MASTER EN SCIENCES INFORMATIQUES

Etude des générateurs de nombres pseudo-aléatoires implétentés dans la technologie d'identification par radiofréquence

Simon, M

Award date:
2012

Awarding institution:
Universite de Namur

Facultés Universitaires Notre Dame de la Paix, Namur

Faculté d'informatique

Année académique 2011-2012

Étude des générateurs de nombres pseudo-aléatoires implémentés dans la technologie d'identification par radiofréquence

Martin SIMON

Mémoire présenté en vue de l'obtention du grade de master en informatique.

Préface

Il est difficile d'imaginer une application cryptographique bien conçue ne reposant pas sur des nombres aléatoires. Qu'il s'agisse de vecteurs d'initialisation, d'une clé de session, du *salt* utilisé dans une fonction de hachage ou de nonces utilisés dans un protocole d'authentification, l'aléatoire est réellement une des bases d'un système cryptographique. Or, la production d'aléatoire est loin d'être simple et beaucoup de systèmes se contentent d'implémenter une source pseudo-aléatoire pour produire de tels nombres. La technologie assez récente des RFID n'échappe bien sûr pas à la règle, et la tâche est d'autant plus compliquée. D'un côté, les concepteurs considèrent parfois que l'utilisation d'un générateur d'aléatoire plus complexe et plus sûr n'est qu'un "gâchis de portes logiques". De l'autre, les tags RFID passifs ne possèdent pas d'alimentation propre et sont donc dépendants du lecteur qui les alimente. Les générateurs de nombres pseudo-aléatoires implémentés dans les tags RFID sont donc sujets à certaines faiblesses. Le but de ce travail est notamment de définir une procédure permettant d'analyser les générateurs implémentés dans ces tags, et plus précisément d'analyser des nonces produits lors du protocole d'authentification du tag. Ce mémoire vise à analyser les particularités des générateurs pseudo-aléatoires implémentés dans la technologie RFID, et à définir une méthode de test claire de ceux-ci. Cette méthode est d'ailleurs appliquée à quelques générateurs existant afin de la valider. Une autre approche de ce travail met en œuvre une série d'expériences physiques permettant éventuellement de constater une variation dans le fonctionnement du générateur de nombres pseudo-aléatoires d'un tag RFID.

Avant-propos

Je voudrais remercier les personnes suivantes pour l'aide qu'ils ont pu m'apporter tout au long de ce travail :

Mon maître de stage, Gildas Avoine, ainsi que l'un de ses chercheurs, Benjamin Martin, pour leurs remarques, leurs conseils et la patience dont ils ont fait preuve tout au long de ce projet.

Jean-Noël Colin, mon promoteur, pour m'avoir permis d'effectuer ce projet de recherche, ainsi que pour ses relectures et ses conseils avisés.

David Spôte et Chong Hee Kim, pour leurs connaissances techniques et sans qui certaines expériences sur les tags RFID n'auraient pas été possibles.

Toute l'équipe de l'"Information Security Group" du département d'ingénierie informatique de l'Université catholique de Louvain, pour son accueil chaleureux et grâce à laquelle j'ai pu travailler dans un cadre particulièrement agréable.

Enfin, je remercie tout simplement ma famille, pour le soutien et les encouragements qu'elle a pu m'apporter tout au long de ce projet. Merci également de m'avoir permis de poursuivre mes études jusqu'à aujourd'hui.

Table des matières

Symboles et acronymes

AES	Advanced Encryption Standard
APDU	Application Protocol Data Unit
BBS	Blum Blum Shub
CSPRNG	Cryptographically secure pseudorandom number generator
DES	Data Encryption Standard
EAS	Electronic Article Surveillance
EEPROM	Electrically Erasable Programmable Read-Only Memory
HF	High Frequency
LCG	Linear Congruential Generator
LF	Low Frequency
LFG	Lagged Fibonacci Generator
LFSR	Linear Feedback Shift Register
NIST	National Institute of Standards and Technology
(P)RNG	(Pseudo)Random Number Generator
RFID	Radio Frequency IDentification
RO	Read-Only
RW	Read-Write
UHF	Ultra High Frequency
UID	Unique Identifier
WORM	Write Once, Read Many

1 Introduction

La technologie RFID (Radio Frequency IDentification) [1] consiste à utiliser des ondes radio afin d'échanger des données entre deux appareils : un lecteur et un tag RFID. Il arrive, pour des communications plus sécurisées, que ces tags contiennent un PRNG (Pseudorandom number generator) [2]. L'étude des PRNG implémentés dans les tags RFID n'a jusqu'à aujourd'hui fait l'objet que de très peu de documents. D'un côté, les constructeurs se gardent bien de présenter les PRNG utilisés dans leurs tags et de l'autre, certaines caractéristiques techniques propres à la technologie RFID en compliquent l'analyse. Les seuls travaux relatifs à de telles études proviennent soit d'une analyse physique du tag au cours de laquelle le PRNG a pu être identifié [NESP08], soit de tests statistiques sommaires effectués sur le PRNG de quelques tags [MHCPL11]. Or, ces caractéristiques rendent l'étude de tels PRNG très intéressante, à cause des faibles moyens techniques mis à leur disposition par le tag RFID. De plus, le protocole d'authentification de tels tags repose presque exclusivement sur leur générateur pseudo-aléatoire. Une faiblesse dans celui-ci permettrait facilement à un attaquant de pouvoir procéder à une attaque par rejeu, par exemple, afin de pouvoir s'authentifier auprès du tag. Le but premier de ce travail sera donc de définir une méthode générique permettant de tester la qualité de générateurs de nombres pseudo-aléatoires implémentés dans la technologie RFID. Pour cela, ce document est divisé en 4 parties.

La première dresse l'état de l'art actuel de la technologie RFID. Une large variété de tags est présente sur le marché, et il convient de distinguer les caractéristiques de ceux-ci. Que ce soit au niveau de leur source d'alimentation, de leur fréquence, de leur mémoire ou de leurs capacités de calcul, ceux-ci ont des modes de fonctionnement bien différents, et il s'agit donc de définir les limites des caractéristiques des tags étudiés dans la suite de ce document. Une présentation du matériel utilisé est ensuite faite, décrivant les lecteurs et les librairies ayant permis de communiquer avec les tags RFID. Cette première partie est clôturée par une description des tags dont le PRNG est analysé dans la suite du document. Ceux-ci sont au nombre de trois et sont tous fabriqués par la société NXP. Le premier type de tag, la Mifare Classic, sert de référence à l'analyse et à l'élaboration de la méthode de test. En effet, celle-ci a pu être analysée par le passé [NESP08] et constituera une bonne base de travail.

La seconde partie de ce document introduit les générateurs de nombres pseudo-aléatoires. Après avoir abordé les différences existant entre l'aléatoire et le pseudo-aléatoire et les différents problèmes inhérents à la production d'aléatoire, cette partie s'intéresse aux différentes caractéristiques liées aux générateurs pseudo-aléatoires proprement dits. Ensuite, une présentation

1. Identification par radiofréquence
2. Générateurs de nombres pseudo-aléatoires

de quelques générateurs connus et utilisés est effectuée, en abordant pour chacun leurs qualités et leurs faiblesses. Finalement, quelques méthodes permettant de tester ces générateurs pseudo-aléatoires sont abordées. Celles-ci consistent à appliquer une série de tests statistiques sur une séquence produite par le générateur afin de déceler certaines faiblesses.

La troisième partie s'intéresse aux générateurs de nombres pseudo-aléatoires implémentés dans la technologie RFID. La technologie RFID impose trois caractéristiques majeures à la génération d'aléatoire : une génération en continu, une réinitialisation de la graine et une imprécision liée au lecteur. De ces caractéristiques découle une méthode d'analyse particulière. L'exemple du PRNG de la Mifare Classic servira de base à l'élaboration d'une procédure impliquant cinq tests semblant pertinents dans la recherche de faiblesses de tels générateurs. Cette procédure de test sera appliquée sur les trois tags présentés dans la première partie. Enfin, cette partie présente une série de commentaires ayant été tirés au vu des résultats de l'application de la procédure sur les trois tags précédents.
Le fil conducteur de cette partie est donné par 2 questions auxquelles ce travail tente de donner des réponses :

- Est-il possible de retrouver les faiblesses présentes dans le PRNG de la Mifare Classic sans pour autant devoir effectuer un long travail de rétro-ingénierie ?
- Est-ce que la société NXP, fabricante de la Mifare Classic, a réalisé les mêmes erreurs dans la conception des PRNG de ses autres tags, que celles présentes dans le PRNG de la Mifare Classic ?

La quatrième partie présente une série d'expériences visant à modifier le comportement du générateur d'un tag RFID de type Mifare Classic. Trois tests seront effectués lors de ces expériences afin de déceler un éventuel dysfonctionnement du générateur du tag. Les trois tests se focaliseront sur la validité des nombres renvoyés par le tag, sur la vitesse de génération du PRNG, ainsi que sur la graine de celui-ci. Les trois expériences appliquées au tag seront un tir de laser sur celui-ci, un refroidissement via une bombe aérosol réfrigérante ainsi qu'un refroidissement plus important du tag grâce à l'utilisation d'azote liquide. Le seul but de ces expériences est de tenter d'identifier des facteurs pouvant influencer le PRNG d'un tag RFID.

Première partie
État de l'art de la RFID

2 Généralités

Une large variété de dispositifs RFID existe sur le marché. Cette section donne tout d'abord une rapide présentation générale de la technologie RFID. Ensuite, les principales différences existants entre les tags sont abordées. Les tags se différencient en effet par des critères tels que leur source d'alimentation, leur fréquence, leur mémoire ou leur capacité de calcul [Fin10]. Il est également important de spécifier les caractéristiques des tags utilisés dans la suite du document.

2.1 Définitions et historique

La RFID (acronyme pour Radio Frequency Identification) consiste à utiliser des champs électromagnétiques et des ondes radio pour obtenir des données sans fil grâce à des étiquettes (ou tags ou encore transpondeurs) RFID. Ces tags sont de petits appareils permettant l'identification d'objets ou d'individus par des machines. La technologie RFID a la particularité de fonctionner à distance, sur le principe suivant : un lecteur émet un signal radio et reçoit en retour les réponses des tags qui se trouvent dans son champ d'action. Un tag RFID contient un microcircuit et une antenne, qui lui permet d'envoyer et de recevoir des données, grâce à un lecteur RFID. Une large variété de tags existe, permettant des applications très variées, allant d'un tag très simple permettant l'identification animale, à des tags plus complexes utilisés par exemple comme passeport biométrique.

Historique

Le premier exemple d'identification par radio fréquence date de la seconde guerre mondiale, et est lié au développement de la radio et du radar. L'identification par fréquence radio était utilisée pour déterminer si un avion rencontré était un avion appartenant au même camp. Néanmoins, le développement de la technologie RFID n'a réellement pris son essor que pendant les années 1970. Depuis ces années, cette technologie est de plus en plus utilisée, pour remplacer progressivement d'autres techniques d'identification telles que les codes barres ou les cartes à bande magnétique.

C'est Charles Walton qui est considéré comme l'inventeur de la RFID, grâce à plusieurs brevets déposés à partir de 1971 [Suh04]. Le tout premier brevet à utiliser le terme "RFID" fut déposé en 1983. Une des premières applications commerciales de la RFID est l'identification du bétail en Europe. Dès le début des années 1980, plusieurs entreprises commencent à fabriquer des tags RFID. Pendant de longues années, la technologie RFID a connu un développement limité, à cause du prix des tags et de leur encombrement. Aujourd'hui, grâce aux progrès de miniaturisation et à une baisse des coûts de production des tags, ceux-ci sont

vendus par milliards chaque année. La figure 1 présente l'évolution du nombre de tags et de lecteurs vendus dans le monde entre 2002 et 2006, ainsi que le prix de ceux-ci.

	2002	2003	2004	2005	2006
Ventes d'étiquettes (en millions d'unités)	150	500	1 200	3 500	15 000
Prix unitaire des étiquettes pour les utilisateurs des plus grands volumes (en dollars)	0,30	0,25	0,15	0,10	0,05
Ventes de lecteurs (en millions d'unités)	0,1	0,3	0,6	1	2

Source : AMR Research et PwC Consulting (2002)

FIGURE 1: Vente et prix des tags RFID dans le monde entre 2002 et 2006

2.2 Applications

La technologie RFID se retrouve impliquée dans une large variété d'applications. En voici une liste non-exhaustive :

- Distribution : suivi et traçage des produits.
- Contrôle d'accès : Identification de personnes ayant accès à un bâtiment par exemple.
- Passeports électroniques (ou biométriques).
- Titres de transports dans les transports en commun (ex : MOBIB).
- Paiement sans contact (ex : Bpaid).
- Billeterie.
- Clé de voiture sans contact.
- Identification animale.

La figure 2 présente 3 applications courantes de la technologie RFID.

(a) Titre de Transport

(b) Billetterie

(c) Identification animale

FIGURE 2: Exemples d'applications des RFID

Sources : (a) Belga, (b) liquidx.net, (c) rfid-smartcard.com

2.3 Source d'alimentation

Les tags RFID peuvent être divisé en 3 catégories, en fonction leur source d'alimentation [MW06] :

Tags passifs

On distingue trois catégories de tags en fonction de leur source d'alimentation. La première, celle qui sera analysée dans la suite de ce document, est la catégorie des tags dits passifs. C'est-à-dire qu'ils ne possèdent pas de source d'énergie interne et doivent donc être alimentés par une source d'énergie extérieure. Toute l'énergie qu'ils consomment (que ce soit pour les calculs internes ou la transmission de données) provient du champ électromagnétique du lecteur, qui alimente tous les tags présents dans son champ. La majorité des tags RFID sont issus de cette catégorie, c'est notamment le cas des tags liés à une simple identification, qui ne nécessitent pas d'être alimentés en permanence.

Tags actifs

La seconde catégorie, celle des tags actifs, concerne tous les tags qui possèdent une batterie interne utilisée pour les calculs et les transmissions de données. Les tags actifs peuvent émettre des données de manière autonome. Grâce à leur batterie, ils possèdent de meilleures portées, de meilleures capacités de calcul et des mémoires plus importantes. Par contre, ils ont une espérance de vie plus courte, sont plus gros et plus chers à produire que les tags passifs. Une batterie peut permettre à un tag actif d'être alimenté pendant une dizaine d'années au maximum.

Tags semi-actifs

La troisième catégorie est celle des tags semi-actifs, tags qui ont également une batterie, mais celle-ci n'est utilisée que pour les calculs internes ou le stockage de données, l'énergie nécessaire pour les transmissions provenant toujours du champ électromagnétique du lecteur. Ce type de tags est utilisé lorsque des calculs permanents sont nécessaires, indépendamment des opérations de lecture/écriture du lecteur.

2.4 Fréquence et portée

Suivant les fréquences et les puissances d'émission utilisées, les tags ont des performances et des coûts de fabrication très différents qui conditionnent leur usage. De plus, la législation interdit certaines plages de fréquences pour ne pas perturber d'autres types de transmission (militaire notamment). La fréquence des technologies RFID se cantonne donc aux fréquences ISM (pour "Industrial Scientific and Medical"). On retrouve 4 plages de fréquence principales pour la technologie RFID :

– Low Frequency (LF) : Tags à basse fréquence (125-135 kHz). Les tags basse fréquence présentent une portée de lecture courte et une vitesse de lecture lente. Les applications

classiques de cette fréquence sont le contrôle des accès, l'identification des animaux, le contrôle de stock et les systèmes antivol de véhicules. La portée est généralement de quelques centimètres.
- High Frequency (HF) : Tags à haute fréquence (13.56 MHz). La portée varie de quelques centimètres à quelques décimètres et la vitesse de lecture est supérieure à celle des tags basse fréquence. Les applications classiques utilisant cette fréquence incluent le contrôle des accès et des titres de transport.
- Ultra High Frequency (UHF) : Tags à ultra haute fréquence (860-960 MHz). Cette fréquence est fréquemment choisie pour les applications de distribution et de logistique. La portée de lecture peut atteindre quelques mètres.
- Microwave : Tags micro-ondes (2.45 GHz). Portée de quelques mètres.

Il existe des tags fonctionnant à d'autres fréquences, mais ces catégories sont les plus utilisées pour les tags RFID [Ser05]. Une description plus précise des différentes fréquences peut par exemple être trouvée dans [Fin10]. L'ISO 18000 (RFID Air Interface family of standards) propose une standardisation de ces fréquences [MW06]. La suite de ce document ne présente que les tags HF, pour leur maturité, leurs usages et leur grande diffusion.

2.5 Mémoire

Capacité

Les tags RFID possèdent au minimum quelques bits permettant de stocker l'identifiant ou UID du tag. Cet identifiant est unique et généralement choisi par le constructeur. L'utilisateur ne peut donc pas le modifier. Il existe cependant quelques exceptions : certains tags ont en effet un UID aléatoire (par exemple, celui du passeport électronique) qui est modifié à chaque fois que le tag est initialisé. De plus, il existe certains tags dont l'UID est modifiable. Certains tags RFID possèdent également une mémoire additionnelle. Généralement, les tags passifs possèdent entre 64 bits et 1 KB de mémoire. Les tags passifs ont souvent plus de mémoire, entre 16 octets et 128 KB. Signalons aussi l'existence de tags ne possédant qu'un seul bit de mémoire (tags Electronic Article Surveillance (EAS)) : c'est peu mais suffisant pour protéger les biens dans les magasins ou les entreprises, par exemple.

Type

On trouve trois types principaux de mémoire dans la technologie RFID :
- EEPROM (Electrically Erasable Programmable Read-Only Memory) : consomme beaucoup d'énergie lors des opérations d'écriture. De plus, le nombre de réécritures possibles est limité (entre 100.000 et 1.000.000). C'est la mémoire actuellement la plus largement répandue dans les tags RFID.
- FRAM (Ferromagnetic Random Access Memory) : Consomme beaucoup moins d'énergie que l'EEPROM (de l'ordre de 100 fois). Le temps d'écriture est également réduit par rapport à l'EEPROM. Néanmoins, des problèmes de fabrication ont retardé sa diffusion sur le marché.

– SRAM (Static Random Access Memory) : Largement utilisée dans les systèmes à micro-ondes. Nécessite une source d'énergie permanente pour la rétention de données.

Accès

On distingue 3 types d'accès différents pour les tags RFID :

– Read-Only (RO) : Mémoire accessible uniquement en lecture. Les données contenues dans ce type de mémoire sont écrites de manière permanente par le constructeur lors de la fabrication du tag. C'est par exemple généralement le cas de la mémoire réservée à l'identifiant unique du tag (UID), qui ne sera accessible qu'en lecture.
– Read-Write (RW) : Mémoire accessible en lecture et en écriture. Ce type de mémoire pourra être lu et modifié par n'importe quel utilisateur capable de communiquer avec le tag. Des questions d'intégrité de données se poseront naturellement pour ce type de mémoire.
– Write Once, Read Many (WORM) : Type de mémoire ne pouvant être modifié qu'une seule fois par l'utilisateur. Une fois réécrite, la mémoire ne sera plus disponible qu'en lecture.

2.6 Capacités de calcul

Les tags RFID peuvent posséder plus ou moins de puissance de calcul. Certains tags n'ont par exemple aucune capacité de calcul et ne peuvent que stocker de l'information. D'autres permettent de réaliser des opérations logiques simples (vérifier un mot de passe par exemple). D'autres enfin sont capables de réaliser de véritables opérations cryptographiques. On distingue :

– Cryptographie symétrique : actuellement utilisée principalement sur les tags HF et LF. La plupart du temps, la cryptographie repose sur DES, AES, ou un algorithme propriétaire.
– Cryptographie asymétrique : actuellement implémentée uniquement dans des tags HF.

2.7 Authentification

La technologie RFID est de plus en plus utilisée dans des applications nécessitant l'utilisation de mécanismes de sécurité contre différentes attaques. De ce fait, des mécanismes d'authentification doivent être mis en place dans la technologie RFID afin de permettre au tag de vérifier l'identité du lecteur tentant d'accéder à ses données. Les explications des deux mécanismes d'authentification des sections 2.7.1 et 2.7.2 proviennent notamment de [Fin10] et [MVO96].

2.7.1 Authentification symétrique mutuelle

Ce mécanisme d'authentification schématisé à la figure 3 est basé sur l'authentification three-pass (ISO 9798-2 "nonce based mutual authentication") dans laquelle chacun des 2 participants vérifie la connaissance de l'autre d'une clé secrète. Le protocole d'authentification

FIGURE 3: Procédure d'authentification mutuelle entre un lecteur et un tag

Source : [Fin10]

débute par l'envoi d'une commande GET_CHALLENGE du lecteur vers le tag. Un nombre aléatoire R_A appelé nonce est alors généré par le tag et envoyé au lecteur. Le lecteur génère ensuite un nombre aléatoire R_B. En utilisant l'algorithme de chiffrement e_K commun et lié à la clé commune K, le lecteur va chiffrer un bloc de données comprenant les 2 nombres aléatoires ainsi que certaines données de contrôle. Il va ensuite envoyer ces données chiffrées ($Token1$) au tag.

$$Token1 = e_K(R_B||R_A||B)$$

$Token1$ est ensuite déchiffré par le tag qui va donc retrouver R_B et R_A, et comparer la valeur de R_A retrouvée via $Token1$ avec le nombre aléatoire qu'il a généré précédemment. Si les deux nombres correspondent, le tag a la confirmation que les 2 clés correspondent. Le tag va alors inverser R_A et R_B dans la réponse qu'il va envoyer au lecteur afin de lui prouver sa connaissance de K. Le bloc $Token2$ transmis par le tag au lecteur est donc :

$$Token2 = e_K(R_A||R_B)$$

Le lecteur va alors vérifier que le nombre aléatoire qu'il a généré pour former $Token1$ est bien semblable au nombre R_B résultant du déchiffrement de $Token2$. Si c'est le cas, le lecteur aura lui aussi la confirmation que les 2 clés correspondent et l'authentification mutuelle sera terminée, chacune des 2 parties ayant pu prouver la connaissance de l'autre de la clé. Cette authentification présente les avantages suivants :

- La clé n'est jamais transmise lors de l'authentification, ce sont seulement des nombres aléatoires chiffrés qui sont transmis.
- N'importe quel algorithme de chiffrement peut être utilisé.

– Une clé de session peut facilement être calculée à partir des nombres aléatoires générés, ce qui permet de sécuriser les transmissions de données futures.

2.7.2 Authentification utilisant des clés dérivées

Un inconvénient de la méthode présentée en 2.7.1 est que pour une même application, tous les tags doivent avoir la même clé secrète. Pour des applications impliquant un grand nombre de tags (par exemple, un système de contrôle de titres de transport pour un réseau de transports en commun, qui implique des milliers de tags), cela peut représenter un risque potentiel. La faible probabilité que la clé soit découverte compromettrait en effet tout le système. Une amélioration peut donc être apportée au mécanisme d'authentification précédent en utilisant des clés différentes lors de l'authentification pour chaque tag. Pour ce faire, l'UID (identifiant unique du tag) du tag va être lu lors de l'authentification. Le lecteur et le tag (et l'ensemble des autres tags de l'application) partagent toujours une clé commune K_M qui est la master key. A partir de cette master key et de l'UID du tag, une clé K_X va maintenant être générée (dérivée). Cette clé K_X sera utilisée lors de l'authentification, en suivant le même protocole qu'en 2.7.1. L'authentification utilisant des clés dérivées est représenté sur la figure 4.

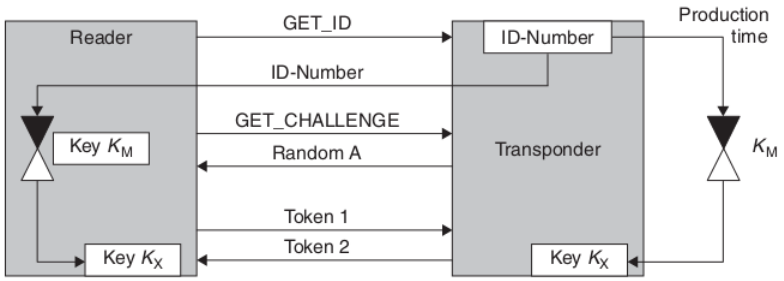

FIGURE 4: Procédure d'authentification par clés dérivées

Source : [Fin10]

2.8 Standards

La technologie RFID s'appuie sur un grand nombre de standards (pas moins de 45 standards en relation avec les RFID sont présentés dans [Fin10]). Néanmoins, ceux-ci ne couvrent cependant aucun mécanisme de sécurité. Les standards impliqués dans la technologie RFID proviennent également de plusieurs organisations (ISO, EPC, ETSI, FCC,...). Ce document ne présente donc que quelques standards utilisés dans les tags HF.

2.8.1 ISO 7816

La norme 7816 a été créée dans un premier temps pour le besoin des smartcards (cartes à puce avec contact). Ce n'est donc pas une norme propre à la technologie RFID. Néanmoins, cette norme, et plus particulièrement sa partie 4 traitant des commandes permettant de communiquer avec les cartes à puce, spécifie la commande à envoyer à la carte pour réaliser diverses opérations. La commande GET CHALLENGE, par exemple, est particulièrement utile dans le cadre de ce travail, car elle permet d'interroger le tag, qui va renvoyer un challenge (*nonce*). La partie 4 de l'ISO 7816 définit la structure (CLA, INS, P1, P2, Data Field) de la commande qui doit être envoyée au tag (voir section 2.8.1).

APDU

Une APDU (pour Application Protocol Data Unit) est l'unité de communication entre un lecteur et un tag RFID (ou une smartcard). La structure d'une APDU est définie dans l'ISO 7816-4. Il en existe 2 types : les APDU de commande (envoyées depuis le lecteur vers le tag) et les APDU de réponse (envoyées par le tag au lecteur). Conformément à l'ISO 7816, la structure d'une APDU de commande est la suivante :

Nom du champ	Longueur (octets)	Description
CLA	1	Type de commande
INS	1	Instruction (ex : "write data")
P1-P2	2	Paramètres de l'instruction
L_c	0, 1 ou 3	Longueur du champ *Data Field*
Data Field	L_c	L_c octets de données
L_e	0, 1, 2 ou 3	Longueur maximale attendue de la réponse

La structure d'une APDU de réponse est la suivante :

Nom du champ	Longueur (octets)	Description
Response	L_r (L_e max)	Bytes contenant la réponse
SW1-SW2	2	Statut de la commande (90 00 indique un succès)

2.8.2 ISO 14443

L'ISO 14443 définit un standard de communication RFID haute fréquence et ses protocoles de communication. Cette norme décrit 2 types de tags : les tags de type A et les tags de type B. Cette norme est divisée en 4 parties :

- Partie 1 : Couche physique.
- Partie 2 : Types de modulation et de codage.
- Partie 3 : Processus d'initialisation et d'anti-collision.

– Partie 4 : Protocole de transmission utilisable pour le type A et le type B qui permet de transférer les commandes définies dans la norme ISO 7816.

Les principales différences entre le type A et le type B se situent au niveau de la modulation et du codage (seconde partie) et des mécanismes d'initialisation et d'anti-collision. Certains tags n'implémentent cependant pas toute la norme iso 14443. La Mifare Classic de NXP, par exemple, se contente d'implémenter les 3 premières parties de la norme et d'utiliser son propre protocole de transmission.

2.8.3 ISO 15693

L'ISO 15693 définit un standard de communication RFID haute fréquence et ses protocoles de communication. Les principales différences avec la norme 14443 se retrouvent dans les techniques de modulation et dans la puissance nécessaire au tag pour fonctionner. Un tag de type 15693 n'a pas besoin d'autant de puissance qu'un tag de type 14443. Par contre, la vitesse de communication est plus importante pour un tag de type 14443. La conséquence de ces différences se manifeste dans les faits par une plus grande portée pour l'ISO 15693 (10 cm maximum pour la 14443, 1m pour l'ISO 15693).

La majorité des tags RFID haute fréquence (13,56 Mhz) utilise une de ces deux normes : ISO 14443 ou ISO 15693.

3 Matériel utilisé

3.1 Lecteurs

Une large variété de lecteurs RFID existe sur le marché, supportant chacun certaines normes, et de ce fait, permettant de communiquer avec certains tags. Les lecteurs suivants ont notamment été utilisés :

ACR122U

Le lecteur ACR122U (présenté à la figure 5) est un lecteur RFID haute fréquence fabriqué par ACS et basé sur le chip PN531 de NXP. Ce chip supporte tous les tags de type ISO 14443 A et B. Il supporte également tous les tags Mifare. Néanmoins, il ne supporte pas les tags de type 15693. L'ACR122U est compatible sous Linux, Windows et MacOSX. La particularité de l'ACR122U est que les APDU envoyées au tag doivent être encapsulées dans des "pseudo-APDU" propres au lecteur.

FIGURE 5: Lecteur ACR122U "Touchatag"

Source : giiks.com

SCL3711

Le lecteur SCL3711 (figure 6) est un lecteur RFID haute fréquence fabriqué par SCM et basé sur le chip PN533 de NXP. Ce chip supporte tous les tags de type ISO 14443 A et B. Il supporte également tous les tags Mifare et Felica. Il ne supporte par contre pas les tags de type 15693.

Omnikey CardMan 5321

Le lecteur Omnikey 5321 (figure 7) est un lecteur RFID haute fréquence fabriqué par HID et basé sur le chip CL RC632 de NXP. Ce chip supporte tous les tags de type ISO 14443 A et B. Il supporte également tous les tags Mifare. Contrairement à l'ACR122U et au SCL3711, l'Omnikey 5321 supporte la norme ISO 15693.

FIGURE 6: Lecteur SCL3711

Source : nfcstuff.com

FIGURE 7: Lecteur Omnikey 5321

Source : scardshop.com

3.2 Software

Cette section vise à présenter les différents outils et librairies utilisés afin de communiquer avec les tags RFID. En effet, il arrive fréquemment qu'un outil ne soit spécifique qu'à un ou plusieurs standards et ne permette de communiquer qu'avec certains lecteurs. Le recours à plusieurs librairies a donc été nécessaire.

3.2.1 Scriptor et gscriptor

Scriptor est un outil permettant d'envoyer directement des APDU à un tag RFID ou une smartcard. Il est possible d'envoyer un fichier d'APDU ou de les introduire manuellement. Gscriptor est une interface graphique pour scriptor. Ces 2 outils sont inclus dans le package pcsc-tools.

3.2.2 Pyscard

Pyscard [3] est un module python ajoutant du support pour les smartcards et tags RFID. Il permet d'envoyer des APDU au tag. Pyscard est très simple d'utilisation, multi-support et supporte les 3 lecteurs présentés dans la section 3.1

3.2.3 Libnfc

Libnfc [4] est une librairie open-source permettant les communications NFC (Near Field Communication). La libnfc est écrite en C et son grand avantage est de permettre une communication avec un tag à un plus bas niveau que les APDU. Il est par exemple possible de manipuler le protocole d'anti-collision grâce à la libnfc. Un autre exemple réside dans le fait que la libnfc permet d'obtenir des challenges (nonces) d'un tag Mifare Classic, ce qui n'est pas possible avec une communication de plus haut niveau telle que les APDU. Le point faible de la libnfc réside dans le fait qu'elle n'est pas compatible avec la norme ISO 15693 et qu'elle n'est supportée que par des lecteurs dont le chip est un PN53X.

3.2.4 Librfid

La librfid [5] est une librairie écrite en C permettant de communiquer avec des tags RFID à bas niveau, tout comme la libnfc. Son grand avantage est qu'elle supporte la norme ISO 15693. Par contre, elle est assez mal documentée, n'est plus mise à jour et ne supporte que peu de lecteurs (OpenPCD ou Omnikey 5x21).

3.3 Présentation des tags utilisés

3.3.1 Mifare Classic

Le tag Mifare Classic fait partie de la famille des tags Mifare, développée en 1995 par NXP. La Mifare Classic est un tag de milieu de gamme, entre la très basique Mifare Ultralight et la plus perfectionnée Mifare DESFire. Il existe 3 variantes de Mifare Classic, en fonction de leur capacité de mémoire (mini : 320 octets, 1K et 4K). La sécurité de la Mifare Classic repose sur une clé de 48 bits. La Mifare Classic est un tag respectant partiellement l'ISO 14443 A, car elle utilise un protocole propriétaire à la place du protocole ISO 14443-4 et ne respecte pas le format des trames défini dans l'ISO 14443-3. La mémoire, la communication et le mécanisme d'authentification de la Mifare Classic sont décrits dans les sections suivantes.

3. `http://pyscard.sourceforge.net/`
4. `http://www.libnfc.org/documentation/introduction`
5. `http://openmrtd.org/projects/librfid/`

Mémoire

Les tags Mifare Classic 1K disposent de 1024 octets de mémoire (EEPROM) répartis en 16 secteurs. Chacun de ces secteurs est subdivisé en 4 blocs de 16 octets. Le 4ème bloc de chaque secteur est réservé pour le contrôle d'accès en lecture et en écriture aux blocs. Suivant les conditions et clés définies dans ce bloc, un utilisateur pourra lire et/ou écrire sur les blocs du secteur en fournissant la clé du secteur au tag. Le 4ème bloc contient principalement :

- La clé secrète A (octets 0 à 5).
- Les conditions d'accès aux blocs du secteur (octets 6 à 9).
- La clé secrète B (optionnelle, octets 10 à 15).

La clé A n'est jamais lisible, la clé B est lisible seulement dans certains cas lorsque A est connue.

Le premier bloc du premier secteur est lui un peu différent, car il n'est pas modifiable par l'utilisateur. En effet, il contient les données du fabricant (UID du tag, somme de contrôle,...).

Communication

Lorsqu'un tag entre dans le champ du lecteur, celui-ci attend une requête venant du lecteur. Lorsque le lecteur veut interagir avec un tag, il envoie une requête à tous les tags présents dans son champ. Les tags répondent alors par un Answer To reQuest (ATQA), signalant leur présence. Une fois que le lecteur a au moins un tag présent dans son champ, celui-ci procède au mécanisme d'anti-collision. A la fin de ce mécanisme, le lecteur connait l'UID de tous les tags présents dans son champ et peut communiquer avec eux de manière individuelle.

Mécanisme d'authentification

Le mécanisme d'authentification du tag Mifare Classic repose sur une authentification par challenge-réponse. Étant donné que le lecteur et le tag partagent un secret commun (la clé du secteur), ils vont, au moyen d'un challenge (communément appelé nonce), mutuellement se prouver qu'ils connaissent ce secret. Le nonce garantit ici une protection contre les attaques par rejeu, où l'attaquant pourrait simplement enregistrer une communication entre le lecteur et un tag afin de la rejouer plus tard.

En pratique, l'authentification de la Mifare Classic se déroule en 3 phases [AMS08]. CRYPTO1, l'algorithme propriétaire de la Mifare Classic, est initialisé à l'aide de la clé secrète :

- (2) Le tag envoie au lecteur un challenge n_T de 32 bits.
- (3) Le lecteur envoie ensuite un challenge n_R au tag. Celui-ci est chiffré avec CRYPTO1. CRYPTO1 a été initialisé avec l'UID du tag et n_T. Le lecteur réinitialise alors CRYPTO1 avec n_R et chiffre maintenant n_T
- (4) Grâce à l'UID, n_T et n_R, le tag peut passer CRYPTO1 dans le même état que celui du lecteur. Il peut donc terminer l'authentification mutuelle en renvoyant n_R chiffré au lecteur.

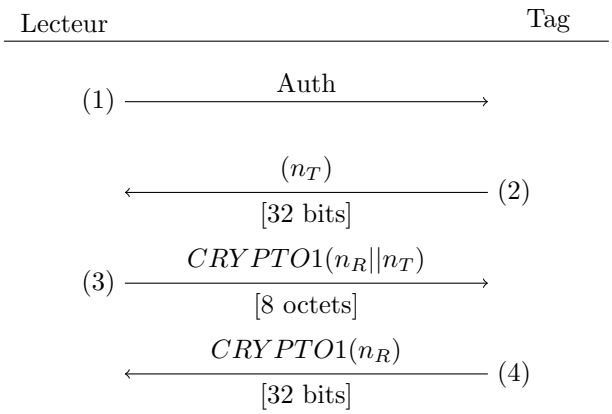

FIGURE 8: Mécanisme d'authentification de la Mifare Classic

Comme il est difficile d'implémenter un nonce véritablement aléatoire sur un tag RFID, ceux-ci ont recours à des fonctions de génération de nombre pseudo-aléatoires, que nous développerons dans la seconde partie de ce document.

Travaux existants

La Mifare Classic a aujourd'hui révélé tous ses secrets. Quatre études dans le courant de l'année 2008 ont permis de dévoiler de nombreuses failles de sécurité présentes dans le tag. La démarche de la première étude, réalisée par Nohl et Plötz, était de réaliser du reverse engineering hardware, en isolant et découpant la puce afin de reconstruire l'algorithme CRYPTO1 [NESP08]. Cette étude a notamment permis de retrouver le générateur pseudo-aléatoire implémenté par la Mifare Classic. La partie III de ce document présente une analyse de ce générateur ainsi qu'une manière pour retrouver le générateur utilisé grâce à cette analyse.

3.3.2 Mifare DESFire

Les tags de type Mifare DESFire appartiennent eux aussi à la famille des tags Mifare de NXP. Contrairement à la Mifare Classic, la Mifare DESFire repose sur l'algorithme de chiffrement DES. La Mifare DESFire repose également sur un système de fichiers basé sur des applications distinctes et des fichiers contenus dans ces applications. Il existe 4 types différents de Mifare DESFire, selon leur taille ou le support de l'ISO 7816. La figure 9 reprend un tableau récapitulatif de ces différences [McL09].

	MF3IC40	MF3IC21–EV1	MF3IC41-EV1	MF3IC81-EV1
Taille de la mémoire	4k	2k	4k	8k
Espace libre	4096 octets	2272 octets	4832 octets	7936 octets
Nombre Max. d'applications	28	28	28	28
Nombre Max. de fichiers par application	16	32	32	32
Crypto	DES, 3DES	DES, 3DES, AES	DES, 3DES, AES	DES, 3DES, AES
Support des commandes de l'ISO 7816	limité	oui	oui	oui

FIGURE 9: Récapitulatif des différents modèles de Mifare DESFire

Authentification

Lorsque le lecteur veut communiquer avec une application du tag (ou un fichier de cette application), il est nécessaire de procéder à une authentification préalable afin de prouver la connaissance de la clé secrète relative à cette application (ou de la clé du fichier) [AM10]. Une fois l'authentification réalisée, le lecteur et le tag établissent une clé de session qui sera utilisée pour les opérations cryptographiques suivantes. Avant de procéder à l'authentification, le lecteur doit sélectionner l'application de la DESFire avec laquelle il veut communiquer. La figure 14 présente le mécanisme d'authentification de la Mifare DESFire décrit ci-dessous.

- (1) Le lecteur envoie au tag la commande relative à l'authentification, avec comme paramètre la clé à utiliser (KeyNo).
- (2) Le tag génère alors un nombre aléatoire n_T de 8 octets qu'il chiffre avec sa clé k_{No} et l'envoie au lecteur.
- (3) Le lecteur déchiffre la valeur reçue avec k_{No} et retrouve n_T. Le premier octet de n_T est ensuite placé à la fin. Ce n_T dont les 8 premiers bits ont été déplacés se nomme n'_T. Le lecteur génère ensuite lui aussi un nombre aléatoire de 8 octets n_R qu'il va concaténer avec $n'T$. Ce nouveau nombre $(n_R||n'T)$ sera ensuite déchiffré (avec k_{no}) et envoyé au tag.
- (4) En chiffrant le nombre reçu, le tag va pouvoir retrouver n'_T et n_R. Il va pouvoir vérifier la valeur de $n'T$. n_T est ensuite lui aussi modifié en n'_T (déplacement des 8 premiers bits de n_T à la fin) et chiffré. $E_{k_{No}}(n'_R)$ est ensuite envoyé au lecteur.
- (5) Le lecteur déchiffre le nombre reçu. Si il retrouve n'_T, les deux parties (tag et lecteur), se sont authentifiées mutuellement.

A la fin d'une authentification réussie, le lecteur et le tag s'accordent sur une clé de session de la forme :

$$\text{Clé de session} = n_{R_{1^{ere}moitie}}||n_{T_{1^{ere}moitie}}||n_{R_{2^{eme}moitie}}||n_{R_{2^{eme}moitie}}$$

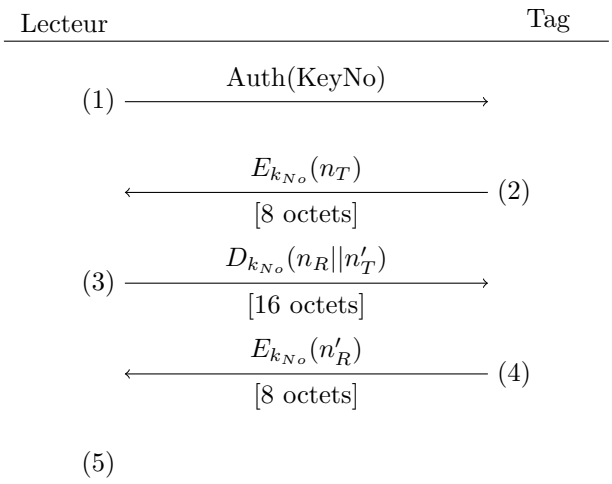

FIGURE 10: Mécanisme d'authentification de la Mifare DESFire

3.3.3 Mifare Ultralight C

La Mifare Ultralight C est un tag low-cost fabriqué également par NXP. Apparue en 2008, la Mifare Ultralight C contient 192 octets de mémoire (EEPROM) divisés en pages de 16 octets (la structure de la mémoire est semblable à celle de la Mifare Ultralight, qui elle ne contient aucun mécanisme de sécurité). Ce tag respecte également les 3 premières parties de la norme ISO 14443 type A. La Mifare Ultralight C permet l'authentification 3DES afin d'obtenir l'accès à des données protégées (authentification mutuelle 3-DES en CBC mode, tout comme pour la Mifare DESFire). Les spécifications de la Mifare Ultralight C peuvent être trouvées dans [NXP11a] ou [NXP09].

Deuxième partie

Introduction aux PRNG

4 Généralités

La sécurité d'un des systèmes d'authentification repose sur la génération d'aléatoire. C'est le cas, par exemple, lors du mécanisme d'authentification présenté dans la section 2.7. Si un tel nombre aléatoire n'était pas utilisé ou s'il pouvait être deviné facilement par un attaquant, une attaque par rejeu, par exemple, pourrait très facilement compromettre ce mécanisme d'authentification. Ce nombre aléatoire doit donc être suffisamment sûr (taille suffisante, le plus "aléatoire" possible,...). Ce chapitre présente donc une introduction à la génération d'aléatoire et aux difficultés et faiblesses rencontrées lors de la génération, afin de permettre la compréhension du chapitre suivant, qui s'intéresse plus spécifiquement à la génération d'aléatoire dans la technologie RFID.

4.1 Définitions

Ces définitions proviennent intégralement d'une traduction de [MVO96].

Générateur de nombres aléatoires

Un générateur de nombres aléatoires (RNG) est un appareil ou mécanisme produisant une séquence de nombres statistiquement indépendants et non biaisés.

Générateur de nombres pseudo-aléatoires

Un générateur de nombres pseudo-aléatoires (PRNG) est un algorithme déterministe qui, étant donnée une séquence réellement aléatoire de longueur k, produit une séquence de longueur strictement supérieure à k, qui semble aléatoire. La séquence passée en entrée est appelée la graine du PRNG.

Distinction entre 2 séquences

Un générateur réussit tous les tests statistiques de temps polynomial s'il est impossible de trouver un algorithme de temps polynomial qui permet de distinguer une séquence produite par le générateur d'une séquence réellement aléatoire avec une probabilité significativement supérieure à $\frac{1}{2}$.

Test du prochain bit

Un générateur réussit le test du prochain bit s'il n'existe aucun algorithme en temps polynomial qui, prenant en entrée les l premiers bits d'une séquence produite s, permet de prédire le $(l+1)^{eme}$ bit de s avec une probabilité significativement supérieure à $\frac{1}{2}$.

Générateur Cryptographiquement sûr

Un générateur est dit cryptographiquement sûr (CSPRNG) si il passe le test du prochain bit avec succès.

4.2 Aléatoire et pseudo-aléatoire

Il faut bien faire la distinction entre une suite de nombres aléatoires et une suite de nombres pseudo-aléatoires. La seconde pourra s'approcher d'un aléa statistiquement parfait mais ne pourra jamais être considérée comme complètement aléatoire. L'utilisation d'un générateur de nombres pseudo-aléatoires permet de se passer d'un générateur réellement aléatoire, plus lourd et compliqué à mettre en place. Un générateur de nombres pseudo-aléatoires permet simplement de ne se servir que d'une petite séquence réellement aléatoire d'une certaine longueur et d'en générer une séquence plus longue afin qu'un adversaire ne puisse pas distinguer si une séquence est réellement aléatoire ou a été générée par le PRNG. Les moyens de générer de l'aléatoire ou du pseudo-aléatoire et leurs caractéristiques sont définis ci-dessous.

4.2.1 Générateurs aléatoires

Un générateur réellement aléatoire nécessite une source naturelle d'aléatoire. Créer un générateur réellement aléatoire est une chose ardue, car la suite de nombres générés ne doit contenir ni biais ni corrélations. De plus, dans le but d'être utilisé en cryptographie, le générateur ne doit pas être observable ou manipulable par un adversaire. Des générateurs basés sur des sources naturelles d'aléatoire pourraient également être influencés par des événements extérieurs, et il faut donc tester régulièrement le bon comportement du générateur.

Générateurs hardware

Un générateur aléatoire hardware permet d'exploiter l'aléatoire survenant dans des phénomènes physiques. En voici quelques exemples [Lom08] :

- Temps écoulé entre l'émission de particules durant la radioactivité. Exemple d'implémentation : HotBits [6].
- Bruit thermique dans l'électronique (d'une résistance, par exemple). Exemple d'implémentation : random.org.
- Mécanique quantique.

Générateurs software

Mettre en place un générateur software est encore plus difficile. De tels générateurs peuvent être basés sur [Lom08] :

- L'horloge du système.

6. http ://www.fourmilab.ch/hotbits

– Le temps écoulé entre des entrées clavier ou souris.
– Différentes valeurs du système d'exploitation.

Le comportement de ces générateurs peut varier considérablement en fonction de la plateforme sur laquelle il est déployé. De plus, il est parfois compliqué de se protéger contre un adversaire tentant d'observer ou de manipuler le générateur. Par exemple, si cet adversaire a une vague idée de la façon dont l'aléatoire est généré, il pourrait deviner le contenu de l'horloge à un certain moment. C'est pourquoi l'utilisation d'une multitude de sources d'aléatoire est une bonne pratique, au cas où l'une de ces sources venait à être compromise. Une autre bonne pratique est d'utiliser une fonction de hachage telle que MD5 ou SHA-1 sur les séquences produites par le générateur.

4.2.2 Générateurs pseudo-aléatoires

La section ci-dessus a présenté les difficultés de l'implémentation d'un générateur véritablement aléatoire. L'utilisation de générateurs pseudo-aléatoires permet donc de générer des nombres semblants aléatoires en passant outre ces difficultés. Un générateur pseudo-aléatoire est défini par une fonction f qui au départ d'une graine s permet d'obtenir une séquence $f(s), f(s+1), f(s+2), ...$ Si le générateur prend successivement le nombre généré comme entrée de la prochaine itération, le générateur produit la suite $f(s), f(f(s)), f(f(f(s))), ...$ Suivant le générateur utilisé, il peut être nécessaire de ne garder que certains bits des différents nombres générés afin de supprimer les possibles corrélations et biais produits par le générateur. Certains générateurs sont cryptographiquement sûrs, tandis que d'autres ne le sont pas. Un générateur non cryptographiquement sûr peut être suffisant pour la plupart des applications ne nécessitant pas une sécurité élevée.

Graine

La graine est le nombre qui initialisera le générateur de nombres pseudo-aléatoires. Cette graine est très importante, car c'est elle qui définit tous les états successifs du PRNG. Obtenir la graine permet de reconstruire l'état interne complet du générateur. Pour des graines semblables initialisant le même générateur, celui-ci produira exactement la même séquence de nombres pseudo-aléatoires. En conséquence, il faut que cette graine provienne d'une source véritable d'aléatoire. De plus, la graine est toujours plus petite que la séquence pseudo-aléatoire générée et est donc plus facile à deviner. La graine est donc le point crucial du générateur et la première source d'attaques.

Périodicité

Un générateur pseudo-aléatoire, au contraire d'un générateur aléatoire, implique la notion de périodicité. Comme un PRNG est défini par une fonction déterministe, celui-ci produira automatiquement le même résultat sur base des mêmes paramètres et du même état. Il arrivera donc un moment où cette fonction prendra des paramètres et un état semblable à

une étape précédente, et le générateur entrera dans un cycle. Le nombre total de nombres aléatoires produits au cours du même cycle est appelé période.

4.2.3 Faiblesses des générateurs

Corrélation

Un générateur (aléatoire ou pseudo-aléatoire) peut être soumis à une corrélation (dépendance entre les bits). Cela signifie que la probabilité que le prochain bit émis par le générateur soit un 1 dépend des autres bits déjà générés par le générateur. Il est assez évident que pour un générateur pseudo-aléatoire classique, le prochain nombre généré sera dépendant des nombres déjà générés par celui-ci. Un moyen d'éviter une corrélation pour un générateur pseudo-aléatoire est d'introduire de l'aléa véritable à chaque étape de génération d'un nouveau nombre pseudo-aléatoire.

Biais

Un générateur (aléatoire ou pseudo-aléatoire) peut être biaisé. Cela signifie que la probabilité que le prochain bit émis par le générateur soit un 1 n'est pas égale à $\frac{1}{2}$. Une technique permettant d'éviter ce biais est la suivante :

Soit un générateur pseudo-aléatoire biaisé mais non corrélé. Etant donné p la probabilité que le prochain bit émis par le PRNG soit un 1 et $1-p$ la probabilité que le prochain bit soit un 0, il suffit de grouper les bits générés par paire afin de remédier au biais. Si la prochaine paire générée est 10, le générateur produira un 1. Si, au contraire, la prochaine paire est 01, le générateur produira un 1. Les paires 00 et 11 sont supprimées. Le générateur sera maintenant non-biaisé.

Autres faiblesses

Le biais et la corrélation ne sont pas les seules faiblesses pouvant être présentes dans un générateur. Voici une liste de quelques faiblesses communes :

- Période trop courte.
- Mauvaise qualité du générateur pour certaines graines.
- Basse qualité de la graine (fixe, pas réellement aléatoire, devinable par un attaquant,...).
- Distribution imparfaite, manque d'uniformité.
- Distribution trop idéale, uniformité trop parfaite.
- Certains bits du nombre produit sont moins aléatoires (Exemple : le bit n°8 du nombre produit est plus souvent un 1 tandis que le n°3 est souvent un 0).

Certaines de ces faiblesses peuvent être assez compliquées à déceler, tandis que d'autres peuvent être visibles à l'œil nu. Afin de pouvoir déceler ces faiblesses, une série de tests doivent être mis en place avant d'utiliser le générateur (voir section 6).

5 PRNG existants

5.1 Générateurs cryptographiquement non-sûrs

5.1.1 Middle-square

La méthode du middle-square (aussi appelée méthode de Von Neumann ou méthode du carré médian) est une des premières méthode ayant vu le jour pour tenter de générer de l'aléatoire [Knu81]. Proposée en 1946 par John Von Neumann (et implémentée pour l'ENIAC), celle-ci n'est pas une bonne méthode permettant de générer de l'aléatoire. Néanmoins, elle est intéressante car elle permet de comprendre facilement le fonctionnement d'un PRNG et d'en découvrir certaines faiblesses. L'idée de la méthode middle-square est de mettre le nombre précédent au carré et de prendre seulement les n chiffres du milieu du carré généré. La figure 11 présente un exemple de cette méthode pour des nombres de $n = 6$ chiffres. Le nombre précédent est 675248. Mis au carré, cela donne 455959861504. de ce carré, on ne retient en sortie que les 6 chiffres du milieu : 959861. 959861 sera également le prochain nombre mis au carré, et ainsi de suite.

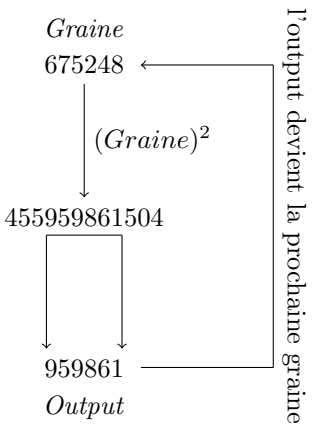

FIGURE 11: Méthode du middle-square

On se rend compte qu'un nombre généré par ce type de générateur est directement dépendant du seul nombre généré précédemment. Néanmoins, cette méthode semble à première vue donner de bons résultats.

Faiblesses

La méthode du middle-square est cependant loin d'être une bonne source de nombres aléatoires, pour plusieurs raisons :

- La période ne sera jamais supérieure à 10^n.
- Si les chiffres du milieu ne sont que des zéros, le générateur ne produira plus que des 0.

– Si la première moitié d'un nombre de la séquence n'est formée que de zéros, le générateur ne produira plus que des nombres tendant vers zéro. Exemple pour $n = 4$: 0099 produit 00009801. Le prochain nombre est 0098, et ainsi de suite. Bien que cela soit facile à détecter cela arrive trop fréquemment dans cette méthode pour qu'elle puisse réellement être utilisable.
– Il existe de petites périodes. Pour $n = 4$ par exemple :
  – 0100
  – 2500
  – 0540-2916-5030-3009

5.1.2 Générateur linéaire congruentiel (LCG)

Les générateurs linéaires congruentiels sont de loin les générateurs les plus populaires et utilisés aujourd'hui. Introduits en 1949 par D.H. Lehmer, ils sont basés sur 4 nombres [Knu81] :

– m, le modulo ; $m > 0$
– a, le multiplicateur ; $0 \leq a < m$
– c, l'incrément ; $0 \leq c < m$
– X_0, la valeur de départ (graine) ; $0 \leq X_0 < m$

La séquence $X_1, X_2, X_3, ..., X_{n+1}$ est ensuite produite par :

$$X_{n+1} = (aX_n + c) \bmod m,\ n \geq 0$$

Par exemple, les paramètres $m = 10, X_0 = a = c = 7$ va produire la séquence :

$$6, 9, 0, 7, 6, 9, 0, 7, ...$$

Cet exemple montre directement que tous les nombres ne sont pas forcément de bons paramètres et qu'il faut apporter un soin particulier au choix de ces paramètres. En effet, la séquence générée n'est pas "aléatoire". Cette séquence est également un exemple parfait de la notion de cycle. Le générateur produira en effet indéfiniment le cycle $6, 9, 0, 7$. La période de ce générateur est donc de longueur 4. Dans cette forme simple, la période du LCG ne pourra jamais être de longueur supérieure à son modulo. Il s'agit donc maintenant de s'intéresser au choix des paramètres.

Choix du modulo

Il faut donc choisir m pour que la période soit assez large, comme celle-ci ne peut être plus grande que le modulo. Le deuxième critère est la vitesse de génération. Il faudra s'assurer que le modulo soit bien choisi dans le but de favoriser par la machine le calcul de $(aX_n + c)$ mod m. Une démonstration de la méthode à utiliser pour effectuer ce choix peut être trouvée dans [Knu81]. Pour résumer, un modulo du type $m = 2^e$ ou $m = 2^e \pm 1$ est un bon choix de modulo.

Choix des autres paramètres

Il s'agit ici de choisir a, c et X_0 afin d'avoir la période la plus grande possible, idéalement maximale (longueur m). 3 conditions sont nécessaires pour obtenir un tel générateur :

- c est premier avec m. $Pgcd(c, m) = 1$.
- Pour chaque nombre premier p divisant m, $(a - 1)$ est un multiple de p.
- $(a - 1)$ est un multiple de 4 si m en est un.

Une remarque est cependant importante à faire : ce n'est pas parce qu'un générateur est de période maximale qu'il produit une séquence aléatoire. L'exemple $a = c = 1$ (générateur $(X_n + 1) \bmod m$) permettra de dissiper les doutes. La notion de potentiel permet alors d'écarter certains LCG de période maximale trop faible. La description de cette notion de potentiel peut être trouvée dans [Knu81].

Un LCG choisi avec de bons paramètres est relativement rapide comparé à d'autres PRNG. La figure 12 présente un tableau comprenant les paramètres de quelques LCG utilisés dans diverses applications.

Source	m	a	c
RANDU	2^{31}	65539	0
Microsoft VisualBasic 6.0	2^{24}	1140671485	12820163
Générateur de Java java.util.Random	2^{48}	25214903917	11
glibc (utilisé par gcc)	2^{31}	1103515245	12345

FIGURE 12: Exemple de quelques LCG utilisés dans des applications répandues

Bien que les LCG soient relativement rapides et qu'ils ne nécessitent que peu de mémoire, ils présentent un inconvénient majeur. En effet, il est possible, moyennant une séquence générée partiellement, de retrouver le reste de la séquence [Hal04].

5.1.3 Lagged Fibonacci Generator (LFG)

Le lagged fibonacci generator se base sur un LCG et en apporte une seule modification importante : le prochain nombre généré X_{n+1} n'est plus généré uniquement par le seul nombre précédent X_n mais par plusieurs nombres précédents (comme dans la séquence de Fibonacci, qui utilise les 2 nombres précédents pour créer le suivant). Un LFG peut être défini sous la forme :

$$X_{n+1} = (X_{n-j} \star X_{n-k}) \bmod m, \text{ avec } 0 \leq j < k$$

Le symbole $\star$ peut être un des opérateurs suivants : addition, soustraction, multiplication ou XOR. La théorie sur ce genre de générateur est assez complexe et les valeurs de j et de k doivent être choisies judicieusement. m est généralement une puissance de 2 ($m = 2^M$), souvent 2^{32} ou 2^{64}. Il est évident que ce type de générateur demande plus de mémoire qu'un simple LCG, car il faut retenir les k derniers nombres générés. Un avantage du LFG est

sa période maximale, qui est plus importante que m. Celle-ci dépend immédiatement de l'opération mathématique utilisée :

- Addition ou soustraction : $(2^k - 1) * 2^{M-1}$
- XOR : $(2^k - 1) * k$
- Multiplication : $(2^k - 1) * 2^{M-3}$

Différents bons choix de j et de k peuvent être trouvés dans la littérature [Knu81]. Par exemple,

$$X_n = (X_{n-24} + X_{n-55}) \bmod m, \text{ avec } n \geq 55$$

de Mitchell et Moore est un bon générateur.

5.1.4 Linear Feedback shift register (LFSR)

Un Linear Feedback shift register (ou registre à décalage linéaire) est un PRNG produisant des bits à partir d'un état interne par un décalage de ceux-ci [Rod]. L'exemple de la figure 13 montre un LFSR de longueur de 8 bits et 4 taps. Les taps sont les positions des bits entrant en compte dans le calcul du prochain bit. Ici, les taps 2, 5, 6 et 8 sont pris en compte dans le calcul du prochain bit. Le polynôme de ce LFSR est donc

$$x^8 + x^6 + x^5 + x^2 + 1$$

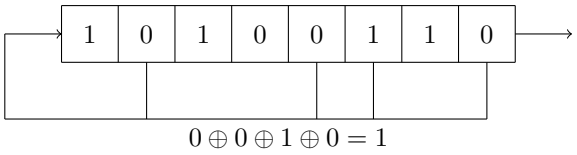

FIGURE 13: Exemple de LFSR

Pour effectuer une itération, le générateur va effectuer une opération XOR sur les valeurs des taps du LFSR. Le prochain bit généré sera donc

$$0 \oplus 1 \oplus 0 \oplus 0 = 1$$

Le générateur va ensuite effectuer un décalage vers la droite. Celui-ci passera de l'état

10100110

à l'état

11010011

Période

La période maximale d'un LFSR est $2^n - 1$ où le LFSR produit tous les 2^n états possibles des bits (sauf le cas où tous les bits sont à 0, et où le LFSR ne changerait jamais d'état) avant de produire un cycle complet. Un LFSR ne sera de période maximale que s'il contient un nombre pair de taps. 2 ou 4 taps peuvent souvent suffire, même pour de grands LFSR. Il existe toujours plusieurs combinaisons de taps permettant d'obtenir une période maximale pour un LFSR de longueur n. En effet, étant donné un LFSR de longueur maximale défini par $x^n + x^A + x^B + x^C + 1$, le LFSR défini par $x^n + x^{n-A} + x^{n-B} + x^{n-C} + 1$ sera également de longueur maximale (exemple : $x^{32} + x^7 + x^3 + x^2 + 1$ et $x^{32} + x^{30} + x^{29} + x^{25} + 1$ seront tous deux de longueur maximale) . Une autre contrainte afin d'obtenir un LFSR de période maximale est qu'il ne doit pas exister de diviseur commun entre tous les taps.

Avantages et inconvénients

Un LFSR est très simple à implémenter en pratique et est également très rapide. Néanmoins, l'utilisation d'un LFSR pour générer de l'aléatoire est déconseillée. En effet, les nombres produits par un tel générateur sont linéaires. De plus, le polynôme d'un LFSR peut être retrouvé avec seulement $2n$ bits générés par le LFSR (algorithme de Berlekamp-Massey). Afin de palier à ces défauts majeurs pour la génération d'aléatoire, 2 techniques peuvent être utilisées : des combinaisons de LFSR où la séquence pseudo-aléatoire produite est la combinaison de plusieurs LFSR ou l'utilisation d'une fonction de filtrage sur la séquence produite par le LFSR (une fonction de hachage, par exemple).

Combinaisons de LFSR

En pratique, l'utilisation de combinaisons de LFSR permet de produire des période plus longues et d'éliminer les dépendances existant entre les nombres générés par un seul LFSR. Le LFSR combiné n'est autre que l'application d'une fonction f booléenne sur n LFSR en entrée : $f(x_1, ..., x_n)$.

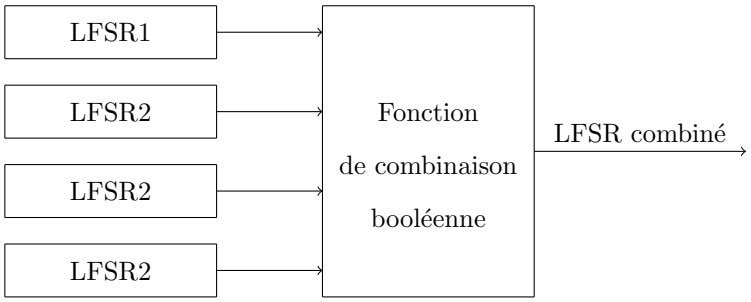

FIGURE 14: Combinaison de LFSR

La sécurité d'une combinaison de LFSR dépend évidemment directement de la fonction booléenne f utilisée[7].

7. http://www.picsi.org/parcours_58_270.html

A chaque itération, tous les générateurs vont produire un nouveau bit. Ces bits vont être combinés suivant la fonction f pour donner le prochain bit du LFSR combiné.

LFSR filtré

Un LFSR filtré n'est rien d'autre qu'un LFSR unique auquel on applique une fonction non-linéaire f sur la sortie. La qualité d'un tel générateur dépend essentiellement du LFSR lui-même et de la fonction de filtrage utilisée.

5.1.5 Mersenne Twister

Le générateur de nombres pseudo-aléatoires Mersenne Twister est un algorithme développé en 1996/1997 par Makoto Matsumoto et Takuji Nishimura [MN98]. Ce générateur est particulièrement réputé pour sa qualité. 2 versions de ce générateur existent, MT11213 et MT19937 qui ont respectivement une période de $2^{11213} - 1$ et $2^{19937} - 1$. Ces périodes sont bien suffisantes pour qu'aucun ordinateur ne puisse produire un cycle sur la durée de vie de l'univers. Étrangement, ce générateur est plus rapide qu'un LCG classique. Néanmoins, ce générateur n'est pas cryptographiquement sûr puisque la connaissance de 624 nombres générés successivement permet de déterminer l'état interne du générateur et ainsi de produire le reste de la suite. Mersenne Twister est un TGSFR (twisted generalised shift feedback register), un type particulier de LFSR. Le Mersenne Twister est notamment le PRNG par défaut de Python, Ruby, et php.

5.2 Générateurs cryptographiquement sûrs

5.2.1 Blum-Blum-Shub

Le générateur Blum-Blum-Shub (du nom de ses inventeurs) est un générateur cryptographiquement sûr qui présente cependant l'inconvénient d'être extrêmement lent. L'algorithme du Blum-Blum-Shub est très simple, et peut être représenté sous la forme [MVO96] :

$$X_{n+1} = (X_n)^2 \bmod M$$

Où M est le produit de p et q, deux grands nombres premiers congruents à 3 modulo 4 ($p - q$ doit être divisible par 3 et par 4). Généralement, on ne prend en sortie de chaque étape que le bit de poids faible (LSB) pour constituer la séquence pseudo-aléatoire.

5.2.2 Yarrow

Yarrow - 160 est un générateur pseudo-aléatoire utilisant l'algorithme 3-DES (E_K). Il est composé de 4 mécanismes [KSF] :

- Un pool d'entropie qui collecte un maximum d'aléa provenant de diverses sources.
- Un mécanisme qui permet de générer une nouvelle clé K périodiquement.

– Un mécanisme de génération utilisé pour générer des séquences pseudo-aléatoires à partir de la clé K.
– Un mécanisme de contrôle qui décide quand il y a lieu de générer une nouvelle clé.

L'algorithme de Yarrow est :

$$C_i = (C_{i-1} + 1) \bmod 2^n \text{ où } C_i \text{ est un compteur}$$
$$R_i = E_K(C_i) \text{ où } R_i \text{ est la sortie du PRNG}$$

Le PRNG Fortuna par exemple, repose sur le même principe que Yarrow, avec AES à la place de 3-DES.

5.3 Erreurs communes

L'histoire des PRNG est parsemée de mauvais algorithmes et des conséquences de leur utilisation. Outre l'exemple du middle-square présenté dans la section 5.1.1, en voici quelques autres.

Modification de bons générateurs

Une des premières erreurs est de se dire que, prenant comme point de départ un générateur relativement bon, il suffit de le modifier un peu pour obtenir un générateur encore meilleur. Par exemple, partant d'un LCG de la forme :

$$X_{n+1} = (aX_n + c) \bmod m$$

qui produit des séquences pseudo-aléatoires relativement bonnes, est-ce qu'un générateur de la forme

$$X_{n+1} = ((aX_n) \bmod (m+1) + c) \bmod m$$

produira des séquences "encore plus aléatoires" ? La réponse est probablement non, et le générateur dérivé produira sans doute des séquences très pauvres en terme d'aléatoire. Comme il n'existe aucune théorie sur le générateur dérivé, celui-ci peut être comparé à un simple générateur $X_{n+1} = f(X_n)$ où f est une fonction choisie aléatoirement et dont le comportement est sans doute moins aléatoire qu'une fonction plus connue et documentée.

RANDU

RANDU est probablement l'exemple le plus connu d'un mauvais générateur. Développé en 1968 par IBM, RANDU est un LCG prenant pour paramètres :

$$X_{n+1} = (65539X_n + 0) \bmod 2^{31}$$

Ces choix en font un générateur très rapide. Néanmoins, la propriété suivante entre trois nombres successifs est insérée à cause des ces paramètres :

$$X_{k+2} = 6X_{k+1} - 9X_k \text{ pour tout } k$$

Cette propriété compromet bien sûr tout le générateur, qui est de ce fait loin d'être aléatoire.

6 Outils existants permettant de tester les PRNG

Bien qu'aucun générateur pseudo-aléatoire ne puisse être dit aléatoire, il est possible de soumettre ces générateurs à des tests afin d'éliminer les générateurs présentant certaines faiblesses. Ces tests sont nécessaires mais pas suffisants. En effet, rien ne dit qu'un générateur réussissant les x premiers tests ne va pas échouer au $x + 1$ème.

6.1 Test du Chi-carré

Le test du chi-carré est un test basique servant de base à toute une série d'autres tests. Voici un petit exemple permettant de l'expliquer très simplement [Knu81]. Soit 144 lancers de deux "vrais" dés. La somme de ces dés est appelée s. La figure 15 présente un tableau des différentes valeurs et valeurs attendues suivant les probabilités des dés pour 144 lancers. On s'attend par exemple à obtenir 4 fois la valeur 2 (double 1), comme la probabilité d'obtenir un double un est de $\frac{1}{36}$. On procède à 3 expérimentations différentes de ces 144 lancers de paires de dés différents. Les valeurs observées sont également reprises dans la figure 15. La question est maintenant de savoir quels dés utilisés lors des 3 observations différentes sont les plus équitables. La réponse intuitive donnée est que les dés utilisés pour l'expérimentation Y_{s2} sont les meilleurs, car ce sont ceux dont la distribution se rapproche le plus de la distribution attendue. Pourtant, le test du Chi-carré va donner d'autres résultats.

Valeur de s	2	3	4	5	6	7	8	9	10	11	12
nombre attendu np_s	4	8	12	16	20	24	20	16	12	8	4
nombre observé Y_{s1}	4	10	10	13	20	18	18	11	13	14	13
nombre observé Y_{s2}	3	7	11	15	19	24	21	17	13	9	5
nombre observé Y_{s3}	2	4	10	12	22	29	21	15	14	9	6

FIGURE 15: Exemple de valeurs attendues et observées pour 144 lancers de 2 dés

La formule utilisée par le test du Chi-carré dans ce cas est la suivante :

$$\chi^2 = \sum(\frac{(o-e)^2}{e})$$

Où o est le résultat observé pour une catégorie, et e le résultat attendu. Appliqué à l'exemple, cela donne pour Y_{s1} :

$$V = \frac{(Y_2-np_2)^2}{np_2} + \frac{(Y_3-np_3)^2}{np_3} + ... + \frac{(Y_{12}-np_{12})^2}{np_{12}}$$

Dès lors, on trouve pour les différentes expérimentations :

$$V_1 = 29 + \frac{59}{120}$$
$$V_2 = 1 + \frac{17}{120}$$
$$V_3 = 7 + \frac{7}{48}$$

La technique du chi-carré consiste alors à vérifier à l'aide d'une table les valeurs obtenues. Il faudra pour cela utiliser une valeur supplémentaire : le degré de liberté. Le degré de liberté n'est autre que le nombre total de possibilités - 1 ($df = k-1$). Dans l'exemple, il y a présence de 11 valeurs différentes possibles ($k = 11$), la ligne de la table analysée sera donc celle où $df = 10$. Dans l'exemple de la figure 15, on remarque que :

Table 5-2
Critical Values of the χ^2 Distribution

df \ p	0.995	0.975	0.9	0.5	0.1	0.05	0.025	0.01	0.005	df
1	.000	.000	0.016	0.455	2.706	3.841	5.024	6.635	7.879	1
2	0.010	0.051	0.211	1.386	4.605	5.991	7.378	9.210	10.597	2
3	0.072	0.216	0.584	2.366	6.251	7.815	9.348	11.345	12.838	3
4	0.207	0.484	1.064	3.357	7.779	9.488	11.143	13.277	14.860	4
5	0.412	0.831	1.610	4.351	9.236	11.070	12.832	15.086	16.750	5
6	0.676	1.237	2.204	5.348	10.645	12.592	14.449	16.812	18.548	6
7	0.989	1.690	2.833	6.346	12.017	14.067	16.013	18.475	20.278	7
8	1.344	2.180	3.490	7.344	13.362	15.507	17.535	20.090	21.955	8
9	1.735	2.700	4.168	8.343	14.684	16.919	19.023	21.666	23.589	9
10	2.156	3.247	4.865	9.342	15.987	18.307	20.483	23.209	25.188	10
11	2.603	3.816	5.578	10.341	17.275	19.675	21.920	24.725	26.757	11
12	3.074	4.404	6.304	11.340	18.549	21.026	23.337	26.217	28.300	12
13	3.565	5.009	7.042	12.340	19.812	22.362	24.736	27.688	29.819	13
14	4.075	5.629	7.790	13.339	21.064	23.685	26.119	29.141	31.319	14
15	4.601	6.262	8.547	14.339	22.307	24.996	27.488	30.578	32.801	15

FIGURE 16: Table des distribution Chi-carré en fonction des degrés de liberté

Source : bio.miami.edu

- $p_{V1} < 0,005$
- $p_{V2} > 0,995$
- $0,5 < p_{V3} < 0,9$

Or,

- Si p_V est inférieure à 0,01 ou supérieure à 0,99, la distribution est considérée comme insuffisamment aléatoire.
- Si p_V est entre 0,01 et 0,05 ou entre 0,95 et 0,99, la distribution est considérée comme suspecte.
- Si p_V est entre 0,05 et 0,1 ou entre 0,90 et 0,95, la distribution est considérée comme presque suspecte.

La conclusion est que les 2 premiers jeux de dés ne sont sans doute pas assez aléatoires (le premier donnant une distribution trop imparfaite et le second donnant une distribution trop parfaite), et que le troisième jeu de dés est le "plus aléatoire". Il faut souvent procéder au moins 3 fois au test du chi-carré. Si au moins deux des trois résultats sont considérés comme

suspects, la séquence observée n'est pas suffisamment aléatoire.

6.2 Outils de tests statistiques

Une bonne manière de tester la qualité d'un générateur de nombres pseudo-aléatoires est le recours à l'utilisation d'une librairie de tests statistiques. Ces librairies contiennent plusieurs tests différents, chacun visant à trouver une faiblesse particulière dans la séquence générée par le PRNG. Un test ayant échoué fournira donc des indications sur la façon d'améliorer le générateur. Pour de bons résultats, il est important de donner en entrée des tests une grande quantité de nombres générés par le PRNG. En effet, chaque test doit être effectué sur plusieurs séquences générées et chaque séquence doit contenir un minimum de bits afin d'être pertinente, et d'apporter assez d'information au test. Il est impossible de valider qu'un générateur est bien aléatoire, quelque soit le nombre de tests passés avec succès, mais il est possible de dire qu'un générateur n'est pas aléatoire, dès lors qu'il échoue à au moins un test. Les 4 sections suivantes décrivent les batteries de tests les plus reconnues et utilisées.

6.2.1 ENT

Ent [8] est un petit programme permettant de tester si une séquence semble aléatoire, en effectuant 5 tests :

- Test d'entropie : donne l'entropie de chaque octet (en bits). Cette valeur est proche de 8 pour un fichier réellement aléatoire. Donne également une valeur de compression du fichier, qui permet de réduire sa taille (plus proche de zéro est la valeur de compression, moins le fichier de bits est compressible et plus il est aléatoire).
- Test Chi-carré : Ce test a été présenté dans la section 6.1.
- Arithmetic mean : C'est simplement la somme de tous les octets (de tous les nombres générés), divisée par la longueur du fichier. Elle doit être proche de 127,5 (255 soit "11111111" divisé par 2) pour un fichier de nombres aléatoires.
- Valeur de Monte Carlo pour π : les valeurs de la séquence permettent de calculer des points successifs qui ont une probabilité de $\frac{\pi}{4}$ de se retrouver dans un cercle de centre $(0, 0)$ et de rayon 1. Le test permet de donner une approximation du nombre π en faisant le rapport des points dans le disque sur le nombre de tirages. Plus l'approximation est proche de π, plus la séquence est aléatoire.
- Serial Correlation Coefficient : Calcule simplement la corrélation entre les octets successifs. Cette valeur doit être proche de zéro pour un fichier réellement aléatoire.

8. `http://www.fourmilab.ch/random/`

6.2.2 DIEHARD

Les tests Diehard [9] ont été développés en 1995 par George Marsaglia. C'est une batterie comprenant 15 tests différents. Les tests de Diehard se sont rapidement imposés comme un standard pour le test de générateurs pseudo-aléatoires, jusqu'à l'arrivée des tests du NIST. A l'exception du *runs test* qui est un test standard (également présent dans les suites du NIST et TESTU01) efficace, tous les tests ont été développés par Marsaglia. Les paramètres des tests sont fixes mais une description de ceux-ci peut-être trouvée pour une réimplémentation avec d'autres paramètres. On retrouve, par exemple :

- Overlapping permutations : Analyse les séquences successives de 5 nombres consécutifs (sur 1 million de nombres). Les 120 arrangements possibles de ces 5 nombres doivent apparaître en fréquence égale dans le million de nombres.
- Binary rank test : Crée des matrices de 31x31, 32x32 et 6x8. Calcule ensuite le rang de ces matrices, en sachant que pour des nombres aléatoires, certains rangs devraient avoir moins d'occurrences que d'autres.
- Différents tests de Monte Carlo (exemple : le craps test où 200 000 parties de craps sont jouées grâce aux fichiers de nombres aléatoires. Le nombre de victoires doit respecter la probabilité de victoire de 200 000 jeux de craps réellement aléatoires).

6.2.3 NIST

La suite de tests statistiques du NIST (National Institute of Standards and Technology) est une série de 15 tests permettant d'identifier les séquences de nombres qui n'ont pas un comportement réellement aléatoire. Cette suite fut développée afin de devenir la suite standard permettant de tester l'aléatoire. Pour cela, ces tests dérivent une p-value pour chaque séquence de bits, qui est en fait la probabilité de la séquence d'avoir été générée par un vrai générateur de nombres aléatoires. Chaque test est réussi si la p-value est supérieure à une certaine valeur (niveau de confiance) α. α sera ici fixée à 0,01. Un test est donc réussi si la p-value de ce test est supérieure à 0,01 (il y a en fait 99% de chances que la séquence n'ait pas été générée par un vrai RNG).
Il est impossible de tester la validité d'un PRNG en ne testant qu'une seule séquence, chaque test doit donc être effectué sur plusieurs séquences du fichier de bits original. Le "Pass Rate" est la proportion de séquences passant le test. Chaque test du NIST requiert un nombre de bits minimum, et certains requièrent des paramètres (Taille des blocs,...). Les tests du NIST sont très compréhensibles car bien documentés. Comme ces tests seront utilisés dans la partie III, une description de chacun d'eux est présentée ici. Une description des algorithmes utilisés par chaque test peut également être trouvée dans [RSN+10].

Test de fréquence

9. `http://www.stat.fsu.edu/pub/diehard/`

Partant du principe qu'une suite aléatoire de bits doit contenir approximativement le même nombre de 1 que de 0, le test de fréquence consiste simplement à déterminer la proportion de 1 présente dans la séquence. Celle-ci doit être proche de $\frac{1}{2}$ pour une séquence aléatoire. Un échec à ce test indique une proportion trop élevée de 1 ou de 0 dans la séquence.

Test de fréquence dans un bloc

Ce test est exactement le même que le test de fréquence, à l'exception qu'il s'applique sur des blocs de M bits. Pour chaque bloc m, le nombre de 1 sera analysé. Celui-ci doit être relativement proche de $\frac{M}{2}$ pour une séquence aléatoire. Pour des blocs de taille $M = 1$, ce test est le même que le précédent.

Runs test

Un run est une suite ininterrompue de bits identiques. Un run de longueur k est donc une suite d'exactement k bits identiques bornée avant et après par un bit de valeur opposée. Le but de ce test est de vérifier si le nombre et la longueur des runs de 1 et de 0 ont le comportement attendu par une séquence aléatoire. En particulier, ce test permet de dire si l'oscillation entre les 0 et les 1 est trop lente ou trop rapide par rapport à une séquence aléatoire.

Test du plus long run de 1 dans un bloc

Ce test permet de déterminer si la longueur du plus long run de 1 à l'intérieur d'un bloc de taille M correspond à la longueur du plus long run de 1 dans une séquence aléatoire. Un test sur le plus long run de 1 est suffisant, le test pour le plus long run de 0 découlant naturellement du premier résultat.

Test du rang des matrices (rank test)

Ce test produit des matrices à partir de la séquence de départ. Le rang de ces matrices sera ensuite vérifié. Le test permet dès lors de vérifier l'absence de dépendance linéaire dans la séquence.

Test de la transformée de Fourier (Spectral test)

Ce test est basé directement sur la transformée de Fourier. Un seuil T est ensuite calculé sur cette transformée. Si la séquence est réellement aléatoire, 95% des valeurs obtenues par la transformée ne doivent pas excéder T. Grâce à l'analyse de la hauteur des pics (valeur dépassant le seuil) de cette transformée, le test est capable de détecter des patterns répétitifs proches les uns des autres.

Test des templates (sans chevauchement)

Le nombre d'occurrences de séquences pré-déterminées de tailles différentes est analysé par ce test, qui permet de détecter si certains patterns de bits sont trop présents dans la séquence générée par le générateur. Ce test est effectué sans chevauchement. C'est-à dire que

pour un pattern de n bits, si celui-ci est trouvé lors de l'analyse de la séquence, le test passe directement au bit suivant le pattern trouvé.

Test des templates (avec chevauchement)

Ce test découle directement du précédent, à l'exception qu'il est effectué avec chevauchement. C'est à dire que contrairement au test sans chevauchement, une fois le pattern trouvé dans la séquence, le test passera simplement au bit suivant et non plus au bit suivant le pattern trouvé.

Test statistique universel de Maurer

Ce test se concentre sur le nombre de bits apparaissant entre des patterns de k bits. Ce test permet de déterminer si la séquence peut être compressée sans perte d'information. En effet, moins il y a de bits entre les patterns, plus la séquence peut être compressée. Une séquence pouvant être significativement compressée n'est évidemment pas considérée comme aléatoire.

Test de la complexité linéaire

Ce test tente de trouver le LFSR le plus petit pouvant représenter l'ensemble de la séquence. Ce test permet de déterminer si la séquence est assez complexe pour être considérée comme aléatoire. Les séquences aléatoires sont représentées par de longs LFSR.

Test des séries

Ce test s'intéresse à la fréquence d'apparition de tous les patterns de m bits. Pour une séquence aléatoire, les différents patterns doivent apparaître avec approximativement la même fréquence. Pour $m = 1$, ce test est équivalent au test de la fréquence. Ce test est effectué avec chevauchement des patterns.

Test de l'entropie approximative

Ce test est comparable au test des séries, à l'exception qu'il compare les fréquences des patterns de m bits avec les fréquences des patterns de $m + 1$ bits. Ce test est également effectué avec chevauchement des patterns.

Test des sommes cumulatives

Une marche aléatoire consiste à faire la somme de tous les bits de la séquence. Si le test rencontre un bit à 1, le test ajoute 1 à la somme. S'il rencontre un bit à 0, il soustrait 1 à la somme. Le test cherche alors à déterminer si au cours de marche aléatoire, la somme des bits s'éloigne trop de 0. En effet, pour une séquence aléatoire, la somme de la marche aléatoire reste proche de 0 à chaque étape.

Test de l'excursion aléatoire

Le test de l'excursion aléatoire divise la marche aléatoire en cycles. Un cycle est une suite d'étapes de la marche commençant et se terminant par 0. Lors de chaque cycle, le

test compare alors le nombre de visites d'un état particulier (par exemple l'état +1) avec le nombre de visites attendues dans une séquence réellement aléatoire. Ce test est divisé en une série de 8 tests, pour chacun des états $-4, -3, -2, -1, +1, +2, +3, +4$.

Test de l'excursion aléatoire (variante)

Ce test est identique au précédent, si ce n'est qu'il s'intéresse au nombre total de visites de chaque état, et plus au nombre de visites de chaque état à l'intérieur d'un cycle. Ce test est effectué sur les états de -9 à +9 (0 exclus).

Résultats

La batterie de tests du NIST produit un fichier de résultat reprenant pour chaque test les différentes catégories de p-values apparaissant, la p-value de l'uniformité de ces catégories, ainsi que la proportions des séquences ayant passé ce test.
Un test est réussi si l'uniformité de ses p-values est correcte (p-value $>$ 0,01) et que la proportion des séquences passant ce test est supérieure à un certain pourcentage (90%). Si l'un de ces critères n'est pas rempli, le générateur n'aura pas réussi le test. Pour des paramètres corrects et un nombre de bits conséquent, tous les tests doivent être réussis.

6.2.4 TestU01

TestU01 est une librairie de tests statistiques développée par Pierre L'Ecuyer et Richard Simard à l'Université de Montréal. TestU01 a pour but de fournir un nombre très large de tests statistiques (il implémente actuellement pratiquement tous les tests connus permettant de tester une séquence pseudo-aléatoire). TestU01 implémente également un grand nombre de générateurs. Pour un usage simpliste, TestU01 rassemble ses tests dans 3 batteries :

- Small Crush (10 tests)
- Crush (96 tests)
- Big Crush (106 tests)

Les tests spécifiques ainsi que leur implémentation appliqués par chaque batterie peuvent être trouvés dans le manuel de TestU01 [LS09].

Troisième partie

Les PRNG dans les tags RFID

7 Introduction

7.1 Limites d'un tag RFID

La partie II a présenté les principales caractéristiques et difficultés inhérentes à la génération d'aléatoire. Or, si certains générateurs cryptographiquement sûrs existent, il semble difficile voire impossible d'implémenter un générateur demandant beaucoup de mémoire et/ou de capacités de calcul sur des appareils aussi simplistes que des tags RFID. Comment, par exemple, générer assez d'aléatoire pour alimenter un générateur de type Yarrow ? Comment calculer rapidement des nombres aléatoires avec une faible capacité de calcul et un générateur Blum-Blum-Shub ? Il est évident que la plupart des générateurs de nombres aléatoires implémentés dans la technologie RFID sont assez simples, à cause des contraintes techniques relatives à cette technologie. Ce document ne s'intéresse qu'aux tags RFID passifs et de haute fréquence.

7.2 Particularités des PRNG utilisés dans les tags RFID

Génération en continu

Les PRNG des tags RFID ayant déjà été analysés dans la littérature (NXP Mifare Classic, HID Iclass et certains tag UHF EPC Class-1 Generation-2) ont tous la particularité de fonctionner en continu. Ils génèrent en fait des nombres aléatoires de manière ininterrompue tant que le tag est alimenté, c'est-à-dire tant qu'il est dans le champ du lecteur (pour un tag passif). Lorsque le lecteur l'interroge, le tag va renvoyer un nombre aléatoire x_0. Il va ensuite continuer de générer des nombres, jusqu'à la prochaine interrogation du lecteur, où le tag renverra un nombre x_{0+t}, t étant le nombre d'itérations du PRNG effectuées entre les deux interrogations du lecteur. Il est impossible pour un tel tag d'obtenir des nombres aléatoires générés successivement par le PRNG du tag.

Bien que le cas n'ait jamais pu être analysé, il n'est pas impossible que pour un tag, le PRNG soit implémenté de manière à ce que celui-ci ne génère des nombres que lorsque le lecteur en fait la requête. Lors d'interrogations successives par le lecteur, le tag renverrait alors les nombres générés successivement par son PRNG. Ce type de générateur est bien sûr plus facile à analyser, et certaines techniques pourraient alors lui être appliquées (l'algorithme de Berlekamp-Massey pour un LFSR, par exemple).

Réinitialisation de la graine

Une autre particularité de la technologie RFID et plus précisément des tags passifs, vient du fait que le générateur utilisé ne démarre que lorsque le tag se trouve dans le champ du lecteur. C'est lors de cette entrée dans le champ du lecteur que le tag initialise la graine de

son PRNG. Dès que le tag sort du champ du lecteur, le PRNG cesse d'être alimenté. Son état est donc perdu et le tag doit le réinitialiser à la prochaine entrée dans le champ du lecteur. Cette particularité est utilisée dans la suite de ce travail afin de tester le caractère aléatoire de cette graine. En effet, les librairies utilisées pour communiquer avec le tag (section 3.2) fournissent des commandes permettant de couper ou d'activer le champ du lecteur. Il est donc possible, en réinitialisant le champ du lecteur (soit en le coupant et en le réinitialisant successivement), de ne générer que les premiers nombres produits juste après l'initialisation du PRNG du tag.

Imprécision du lecteur

Le dernier point à souligner lors de la génération de nombres aléatoires concerne l'imprécision liée au lecteur. En effet, lorsque le lecteur interroge le tag afin d'obtenir un nombre déterminé via un timing précis, le nombre renvoyé par le tag se situera dans un intervalle plus ou moins important. Cette imprécision dépend du lecteur, de l'environnement dans lequel le lecteur interroge le tag et des librairies utilisées pour interroger le tag.

7.3 Exemple : Mifare Classic

La Mifare Classic, présentée à la section 3.3.1, génère lors de l'authentification un nombre aléatoire r_T de 32 bits. Les études précédentes (notamment [NESP08] et [GdKGM+08]) ont permis de retrouver le PRNG utilisé pour générer ce nombre. Il s'agit d'un LFSR de 16 bits de la forme :

$$x^{16} + x^{14} + x^{13} + x^{11} + 1$$

Ce LFSR est de période maximale, donc de période $2^{16} - 1 = 65535$. Les bits 17 à 32 sont composés par le résidu du LFSR, comme présenté sur la figure 17. Le tag effectue alors une

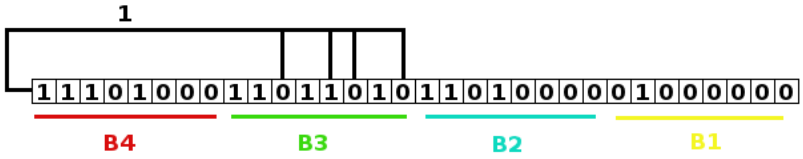

FIGURE 17: LFSR implémenté dans la Mifare Classic

rotation entre les octets, pour renvoyer au lecteur un nombre sous la forme B1||B2||B3||B4. Une constatation évidente est que B1 et B2 permettent directement de reconstituer B3 et B4. En effet, en appliquant 16 itérations du LFSR 16 bits $x^{16} + x^{14} + x^{13} + x^{11} + 1$ prenant comme graine $B1||B2$, la séquence $B3||B4$ est produite.

Nohl, Plötz, Staburg et Evans

Nohl, Plötz, Staburg et Evans sont des chercheurs ayant réussi à effectuer une rétro-ingénierie hardware du tag Mifare Classic. Pour cela, ils ont abordé le problème en découpant la puce du tag en tranches. A partir de photos de ces tranches analysées au microscope, ils ont pu identifier les différentes portes logiques du tag. Cette analyse des portes logiques leur a notamment permis de retrouver le PRNG utilisé par la Mifare Classic. Cette méthode est cependant loin d'être simple à mettre en œuvre, car elle nécessite un lourd matériel.

La procédure détaillée dans la section suivante vise à éviter une analyse hardware du tag afin d'en retrouver le PRNG. Pour cela, cette procédure considère le PRNG comme une *black-box* produisant des séquences de nombres, et lui applique une série de tests afin d'identifier des faiblesses ou le type de PRNG implémenté.

8 Procédure de test du PRNG d'un tag RFID

8.1 Objectif

Cette procédure de test est destinée à identifier le PRNG implémenté dans un tag RFID et à en trouver les faiblesses. Pour cela, une série de tests est appliquée sur un échantillon de nombres aléatoires générés.
Ces tests ont été élaborés à partir des faiblesses rencontrées sur un tag de Mifare Classic et sont valables pour un générateur produisant des nombres aléatoires en continu tant que le tag est présent dans le champ du lecteur. En présence d'un éventuel générateur ne produisant des nombres aléatoires que quand il est interrogé par le lecteur, la procédure peut être simplifiée.

La procédure peut être représentée sous la forme d'un diagramme présentant le cheminement des tests à effectuer. Ce diagramme est présenté à la figure 18 :

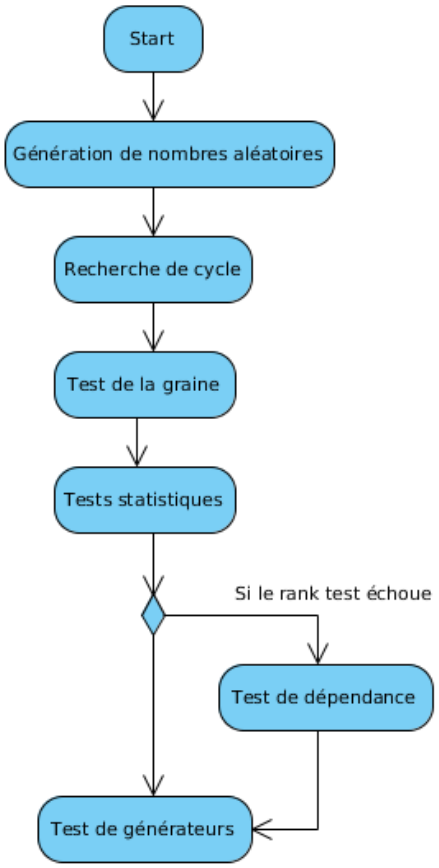

FIGURE 18: Diagramme du cheminement des tests

Les différentes étapes sont expliquées dans la suite de ce document.

8.2 Génération de nombres aléatoires

8.2.1 Description

Cette étape consiste à générer des échantillons de nombres aléatoires selon deux approches différentes. Les résultats sont stockés dans des fichiers pour être utilisés ultérieurement.

- Le premier fichier contient 90 millions de bits générés en continu. La taille de l'échantillon permet d'appliquer tous les tests du NIST. Il permet également de trouver un cycle lorsque la période du générateur est suffisamment petite. En plus de cela, il faut mesurer le temps total nécessaire à la génération de l'échantillon complet. Cette information sera utile pour le test de la section 8.3. Ce fichier est appelé *nombres_non_reinit* dans la suite du document.
- Le second fichier contient 5000 nombres aléatoires séparés à chaque génération par une coupure et une réactivation du champ du lecteur. Cette génération doit se faire dans un environnement minimal (live-cd, pas de réseau,...) afin d'avoir le moins de perturbations possible provenant de l'environnement de test. Par exemple, le simple fait de désactiver le réseau lors de la génération a permis expérimentalement de passer d'une imprécision de 10000 nombres à une imprécision de 4000 nombres pour un tag de type Mifare Classic et l'utilisation de la librairie *libnfc*. Concrètement, une imprécision de 4000 nombres signifie que lors d'une interrogation du tag par le lecteur en vue d'obtenir un nombre précis, le nombre renvoyé le sera dans un intervalle d'itérations du générateur de -2000 à 2000. Ce fichier est appelé *nombres_reinit* dans la suite du document.

Il est difficile de concevoir un outil permettant de générer un tel fichier pour n'importe quel tag. En effet, ceux-ci utilisent des moyens parfois très distincts pour communiquer avec le lecteur, et suivant le lecteur et le tag utilisés, il est préférable d'utiliser l'une ou l'autre librairie.

Le fichier doit se présenter sous la forme suivante pour être utilisable par tous les tests :

- Format ASCII.
- Un nombre aléatoire (sous forme binaire) par ligne.
- Le même nombre de bits par nombre (éventuellement, rajouter des bits à 0 au début du nombre).

8.2.2 Pratique

À titre d'exemple, les programmes *genClassic* et *genClassic_reinit* fournis en annexe permettent de générer respectivement les 2 fichiers via la libnfc pour un tag de type Mifare Classic. La commande à utiliser est la suivante :

```
./genClassic nombre_de_nombres fichier_output
```

Les scripts *geneDESULC.py* et *genULC.c* permettent quant à eux de générer des nombres aléatoires pour la DesFIRE et l'Ultralight C en utilisant respectivement directement des APDU, ou en passant par la libnfc.

8.3 Recherche d'un cycle

8.3.1 Description

Comme précisé dans la section 4.2.2, un générateur de nombres pseudo-aléatoires implique la notion de cycle. Après un certain nombre d'itérations, il reproduit exactement la séquence initiale. Le nombre de ces itérations est appelé *période*.
L'objectif du test est de trouver un cycle dans le fichier *nombres_non_reinit*, afin de déterminer la période du PRNG. Comme le temps pris par le tag pour effectuer un cycle n'est à priori pas connu, un double n'apparaîtra pas de façon certaine dans *nombres_non_reinit*. Dans ce cas, la conclusion est que le générateur a une période trop élevée pour pouvoir être déterminée avec ce test dans un temps raisonnable. La seule information retirée d'un tel résultat est que le générateur a une période plus élevée que le nombre de nombres générés dans *nombres_non_reinit*. Ce test est divisé en deux parties, chacune d'elle permettant de donner une évaluation de la période du générateur. Ces tests sont complémentaires et chacun d'eux permet de confirmer ou d'infirmer le résultat de l'autre.

Test du plus petit double

Le test consiste à tenter de retrouver un nombre aléatoire généré en double dans le fichier *nombres_non_reinit*. En effet, si le test identifie un nombre en double, cela signie que le générateur aura produit au moins un cycle si le générateur ne produit pas plusieurs fois le même nombre dans un cycle. Comme le test tente d'identifier un seul cycle (ou en tout cas le plus petit nombre de cycles), il ne retiendra que le nombre présent en double dans le fichier et dont les deux itérations sont séparées par le plus petit nombre de nombres aléatoires possible.

La suite du test consiste à tenter de donner une évaluation chiffrée de la période grâce au temps pris par le générateur pour produire un ou plusieurs cycles et grâce à la fréquence de celui-ci. La formule *fréquence du générateur* $*$ *temps mis pour générer un ou plusieurs cycles* nous donne un multiple de la période. La fréquence peut être trouvée dans la documentation du tag RFID. Grâce au nombre x de nombres aléatoires générés en y secondes (timer), le temps mis pour générer un ou plusieurs cycles peut également être déterminé. En effet, si z est égal au plus petit écart entre deux nombres aléatoires apparaissant en double dans le fichier, le temps mis pour générer ces cycles est égal à $\frac{y}{x} * z$. Et donc, un multiple de la période du générateur correspond à $frequence * \frac{y}{x} * z$.

Cette méthode est cependant fort imprécise et ne fonctionne que pour certaines implémentations. Néanmoins, pour un générateur non particulier, elle permet de donner une borne supérieure pour la période du générateur (les recherches de générateur peuvent se limiter aux PRNG ayant une période plus petite ou égale).

Test des tirages

Une seconde méthode d'évaluation de la période est également effectuée sur base du fichier *nombres_non_reinit*. Etant donnée, pour un PRNG ne produisant pas 2 nombres semblables lors du même cycle, la probabilité

$$\Pr(b' = b) = \frac{1}{\rho} \text{ avec } \rho \text{ la période du PRNG}$$

de tirer un nombre b' égal à un premier nombre b de référence. En effectuant ce test un grand nombre de fois, il est possible de donner une approximation de la période. En pratique, ce test sera effectué pour chaque nombre du fichier *nombres_non_reinit*. Pour chacun des nombres, il faudra compter le nombres de doubles retrouvés. La fraction $\frac{nombrededoublestrouves}{nombredetestseffectues}$ sera donc une approximation expérimentale de $\frac{1}{\rho}$ et ρ sera évalué à $\frac{nombredetestseffectues}{nombrededoublestrouves}$.

Le test des tirages tente de donner une évaluation précise, tandis que le test du meilleur double peut donner un multiple de la période si plusieurs cycles se sont produits avant de retrouver un nombre en double. Il est donc clair que si la période trouvée par le test du meilleur double est approximativement un multiple du test des tirages, ceux-ci ne sont pas contradictoires. Si au contraire, ces deux tests ne correspondent pas, il est probable que le générateur implémenté dans le tag analysé soit un générateur plus particulier, comme mentionné dans la section 8.3.4.

8.3.2 Algorithme

Test du plus petit double

Le test effectue en premier lieu un quicksort (tri rapide) sur le fichier *nombres_non_reinit* en retenant, pour chaque nombre aléatoire, sa position initiale. Ce tri est un des plus rapides et des plus utilisés en pratique. Ensuite, il parcourt toute la liste maintenant triée des nombres aléatoires. Si, au cours de ce parcours, deux nombres successifs sont égaux et que leur différence de position est plus petite que celle trouvée jusqu'ici, ce nombre devient le nouveau double dont l'écart entre ses occurrences est le plus faible. Ce nombre est retourné à la fin du test. L'écart entre les deux occurrences de ce nombre est également retourné.

Test des tirages

Pour donner l'évaluation de la période, le test initialise deux compteurs à la valeur 0 : C et T. Il parcourt toute la liste triée. A chaque fois qu'il trouve un double lors du parcours de la liste ($liste[n] = liste[n+1]$), C est incrémenté. Dès qu'un double n'est pas trouvé, ($liste[n] \neq liste[n+1]$), la taille de la liste est ajoutée à T. L'évaluation de la période du générateur est donnée, une fois toute la liste parcourue, par $\frac{T}{C}$.

8.3.3 Pratique

Lancer le programme PRNGAnalyzer.jar.
Spécifier le chemin d'accès au au fichier *nombres_non_reinit* et sélectionner le test de la

période.
Le résultat se présente dans le fichier result.txt sous la forme :

```
Double X trouvé, en I et J. Ecart : Y
Estimation de la période : K
Fin du test de la période
```

Où :
- X est le nombre aléatoire trouvé en double avec un écart minimal,
- I et J sont les positions (numéros de lignes) dans le fichier de nombres aléatoires correspondant aux deux occurrences les plus favorables du nombre,
- Y est l'écart entre ces 2 nombres $(J - I)$.
- K est la période estimée par le test des tirages.

Si aucun double n'est trouvé, la première ligne du résultat n'apparaît pas.

Le calcul $\frac{y}{x} * Z$ secondes donne le temps t après lequel le générateur aura fait 1 cycle (ou 2, 3,...). Connaissant maintenant le temps pris par le générateur pour boucler et connaissant la fréquence du générateur, la période maximale est évaluée à $frequence * t$.

8.3.4 Générateurs particuliers

La recherche de cycle n'est cependant pas applicable à tous les générateurs. Premièrement, il sera impossible en pratique de trouver un cycle lorsque le PRNG possède une période très élevée.
Deuxièmement, pour certains générateurs, le fait de trouver un nombre aléatoire en double ne signifie pas pour autant qu'un cycle a eu lieu. Exemple :

- $x_{n+1} = (x_n + x_{n-1}) \bmod m$
- pour $m = 3$ et $x_0 = 0$ et $x_1 = 1$, produit :
- 1, 2, 0, 2, 2, 1, 0, 1, 2,...
- 2 apparaît plusieurs fois à l'intérieur du même cycle

De la même manière, la présence d'une fonction de filtrage à la sortie du générateur produisant le même nombre en sortie pour certaines valeurs différentes de nombres en entrée rendrait également la détection du cycle plus compliquée.

8.4 Test de la graine

8.4.1 Description

Le but de ce test est de vérifier le caractère aléatoire de la graine du PRNG. Ce test s'effectue donc sur le second fichier, *nombres_reinit*. Le test calcule le nombre de nombres aléatoires différents et de collisions multiples se trouvant dans le fichier. Si le nombre de nombres différents est faible et que les 20 nombres les plus présents ont un nombre élevé d'apparitions, il est possible de déduire que la graine du PRNG n'est pas aléatoire. Sinon,

celle-ci n'est pas fixe [10].

Pour donner un ordre de grandeur (très relatif, car cela dépend de la période, à priori inconnue), à partir de 4500 nombres aléatoires différents et d'un maximum de 5 collisions (si la période du générateur est très courte) jusque 5000 nombres aléatoires différents et un maximum de 1 collision multiple, la graine ne peut pas être dite fixe. Si l'on obtient moins de 1000 nombres aléatoires différents et plus de 100 comme maximum d'occurrence d'un même nombre aléatoire, elle peut être dite fixe.
Éventuellement, pour un résultat n'entrant dans aucun de ces critères, générer un second échantillon permettrait de lever l'indécision.
5000 nombres aléatoires permettent d'obtenir des résultats concluants. Générer plus de nombres aléatoires permet d'obtenir un résultat plus fin, mais 5000 était un nombre suffisant dans toutes les expériences réalisées.

8.4.2 Algorithme

Le programme parcourt le fichier *nombres_reinit*. Pour chaque nombre présent dans le fichier, le programme le compare avec une liste initialisée au début du test. Si le nombre est présent dans le fichier, il incrémente le compteur de ce nombre de 1. S'il n'est pas encore présent, il le rajoute en fin de liste en initialisant son compteur à 1. Une fois le fichier parcouru, le test renvoie la taille de la liste ainsi que le tableau des 20 nombres dont les compteurs sont les plus élevés.

8.4.3 Pratique

Lancer le programme PRNGAnalyzer.
Spécifier le chemin d'accès au fichier *nombres_reinit* et sélectionner le test de la graine. Le résultat se présente dans le fichier result.txt sous la forme :

```
Nombre de nombres différents : X
Nombre max d'occurrences d'un nombre : Y fois, pour Z
Tableau des 20 nombres appraissant le plus de fois : T
Fin test du nombre de nombres aléatoires semblables
```

Où :

- X est le nombre de nombres différents trouvés dans le fichier
- Y est le nombre maximal d'occurrences du même nombre dans le fichier
- Z correspond au nombre qui apparaît le plus de fois dans le fichier
- T est un tableau reprenant les 20 nombres ayant le maximum d'occurrences

10. ou elle n'est pas réinitialisée entre chaque coupure du champ, ce qui est improbable, car cela nécessiterait le stockage de la graine dans la mémoire du tag

8.4.4 Particularité

Si le PRNG implémenté dans le tag est connu, il est possible à partir d'un fichier implémentant tous les nombres aléatoires possibles du générateur, de voir si les nombres aléatoires générés sont étalés sur l'ensemble des nombres possibles ou au contraire s'ils se retrouvent dans un faible sous-ensemble des nombres possible. Dans le second cas, la graine ne serait pas aléatoire. Pour ce faire, le programme divise l'ensemble des nombres aléatoires pouvant être générés en x, suivant leur index. Cette division crée donc x catégories contenant des intervalles d'index. Le programme va alors compter le nombre de nombres aléatoires dont l'index apparaît dans chacune des catégories. Si celui-ci est dans le même ordre de grandeur pour chaque catégorie, la graine est considérée comme aléatoire. Si des pics sont présents dans certaines catégories, elle est considérée comme fixe.

Il faut ici spécifier au programme PRNGAnalyzer.jar que le générateur est connu, et lui donner le chemin d'accès au fichier implémentant la suite de tous les nombres aléatoires possibles du générateur. Le résultat se présente dans le fichier result.txt sous la forme :

```
Distribution des nombres :
Entre a et b : X
Entre b+1 et c : Y
Entre c+1 et d : Z
...
```

Où $a, b, c, ...$ représentent les index de début et de fin de catégorie et $X, Y, Z, ...$ le nombre de nombres aléatoires obtenus dans chaque catégorie. Grâce à ces données, un diagramme de la répartition des nombres sur l'ensemble des valeurs possibles du générateur peut être tracé.

Test du chi-carré

Afin de valider l'idée intuitive sur le caractère aléatoire de la graine donné par le diagramme, un test du Chi-carré va alors être effectué sur ces différentes catégories. Le test ne peut pas être effectué directement sur l'ensemble des 5000 nombres. En effet, selon Robert Sedgewick, pour que le test soit valide, il faut que N soit plus grand que $10 * r$ (N étant le nombre d'expériences, et r le nombre de valeurs possibles). Comme il est très compliqué dans un temps raisonnable de générer $10 * r$ nombres, le test va diviser les index des valeurs possibles en 100 catégories différentes. Le test acquiert ainsi un nombre suffisants d'observations par rapport aux valeurs possibles, et devient ainsi valide. Les r différentes valeurs sont équiprobables, tout comme les 100 catégories nouvellement analysées. Comme précisé à la section 6.1, le test sera effectué sur 3 séquences générées, afin d'éviter une éventuelle erreur provenant d'un échantillon pas assez favorable. Le test est effectué sur les 100 catégories. La valeur attendue pour chaque catégorie est donc de $\frac{5000}{100} = 50$. Pour un degré de liberté de 99, l'interprétation du résultat du test est la suivante [11] :

11. `http://econ.clarion.edu/econ222/eagle222/chisquaretable.htm`

– Si $\chi^2 > 134.642$ ou $\chi^2 < 69.230$: insuffisamment aléatoire.
– Si $123.225 < \chi^2 \leq 134.642$ ou $77.046 > \chi^2 \geq 69.230$: suspecte.
– Si $117.407 < \chi^2 \leq 123.225$ ou $81.449 > \chi^2 \geq 77.046$: presque suspecte.
– Sinon ($81.449 \leq \chi^2 \leq 117.407$), la distribution est suffisamment aléatoire.

Au niveau de l'implémentation, le test prend en entrée le tableau des occurrences des cent catégories, et applique la formule du chi-carré en prenant comme nombre attendu $\frac{5000}{100}$. Le résultat est ensuite comparé aux différentes valeurs ci-dessus afin de lui donner une interprétation.

8.5 Tests statistiques

8.5.1 Description

La suite de tests statistiques du NIST a été détaillée dans la section 6.2.3. Afin de pouvoir effectuer tous les tests avec un nombre de séquences suffisant, ceux-ci sont effectués sur le fichier *nombres_non_reinit*.
Les paramètres pour les tests du NIST sont détaillés dans la partie pratique.

Un exemple de fichier de résultats des tests du NIST est donné à la figure 19. Dans ce fichier, le test de fréquence donne comme résultat 2 p-values entre 0 et 0,1, 3 p-values entre 0,1 et 0,2,... la p-value de l'uniformité de ces catégories de p-values est de 0.689019 et la proportion des tests réussis est de 21/21.

```
------------------------------------------------------------------------------
RESULTS FOR THE UNIFORMITY OF P-VALUES AND THE PROPORTION OF PASSING SEQUENCES
------------------------------------------------------------------------------
   generator is </home/gsi/Bureau/NIST_tests/test/32Rand.txt>
------------------------------------------------------------------------------
 C1  C2  C3  C4  C5  C6  C7  C8  C9 C10  P-VALUE  PROPORTION  STATISTICAL TEST
------------------------------------------------------------------------------
  2   3   2   3   2   4   1   0   3   1  0.689019     21/21      Frequency
  0   3   2   3   0   3   1   2   2   5  0.311542     21/21      BlockFrequency
  1   3   0   3   2   2   4   2   3   1  0.689019     21/21      CumulativeSums
  2   2   3   4   0   3   0   4   1   2  0.392456     21/21      CumulativeSums
  3   0   4   2   1   2   2   1   3   3  0.689019     21/21      Runs
  1   2   6   2   3   2   1   1   2   1  0.311542     21/21      LongestRun
  3   2   3   2   0   4   2   0   2   3  0.585209     21/21      Rank
  2   1   2   3   3   3   3   3   1   0  0.788728     21/21      FFT
```

FIGURE 19: Exemple de résultats des tests du NIST

Le tableau de la figure 20 donne une brève description de la faiblesse résultant d'un test échoué :

N°	Test	Erreur
1	Fréquence	Trop de 0 ou de 1
2	Fréquence dans un bloc	Trop de 0 ou de 1 dans un bloc
3	Runs	Trop grand (respectivement petit) nombre de runs, indiquant que l'oscillation entre les 0 et les 1 est trop rapide (lente)
4	Plus long run de 1	Le plus long run de 1 est trop grand/petit dans un bloc
5	Rank	Identification d'une dépendance entre certains bits
6	Fourier	Détection de patterns répétitifs proches les uns des autres
7	Non-overlapping Template	Trop d'occurrences de certains templates sans chevauchement (voir le fichier de résultats pour l'identification du template en question)
8	Overlapping Template	Trop d'occurrences de certains templates avec chevauchement (voir le fichier de résultats pour l'identification du template en question)
9	Universal	La séquence est trop compressible
10	Linear Complexity	Le LFSR pouvant représenter les sous-séquences est trop simple
11	Serial	Les différents patterns de m bits n'apparaissent pas avec une fréquence uniforme
12	Entropie approximative	Les différents patterns de m bits n'apparaissent pas avec une fréquence uniforme en comparaison avec les patterns de $m+1$ bits
13	Sommes cumulatives	Lors d'un random walk de la séquence, la somme s'éloigne trop de 0
14	Random Excursions	Le nombre de visites de l'état 0 lors d'une random walk est trop élevé (faible)
15	Random Excursion Variant	Le nombre de visites de certains états (de -9 à 9 excepté l'état 0) lors d'une random walk est trop élevé (faible)

FIGURE 20: Tableau des interprétations des échecs aux tests du NIST

Afin d'obtenir des indices sur le PRNG implémenté dans le tag RFID, plusieurs générateurs différents ont été testés par la suite du NIST. Cela permet éventuellement, lors de l'analyse du fichier *nombres_non_reinit*, de trouver un générateur obtenant approximativement les mêmes résultats. Des exemples de résultats de tests pour différents types de PRNG sont présentés sur les figures 21, 22 et 23. La figure 21 présente des résultats pour des générateurs faibles, qui produisent un nombre à chaque étape du générateur. Les différents LFSR sont de période

maximale et de polynôme :

- (1) $x^{16} + x^{14} + x^{13} + x^{11} + 1$
- (2) $x^{32} + x^{22} + x^2 + x^1 + 1$
- (3) $x^{64} + x^{63} + x^{61} + x^{60} + 1$
- (4) $x^{16} + x^{14} + x^{13} + x^{11} + 1$ étendu sur 32 bits, c'est-à-dire que les 16 bits de poids faible correspondent à l'application du LFSR avec les 16 bits de poids fort servant de graine.

Test	LFSR 16 bits (1)	LFSR 32 bits (2)	LFSR 64 bits (3)	LFSR 16 bits étendu sur 32 bits (4)
1	90/90	34/90*	22/90*	51/90*
2	0/90*	0/90*	0/90*	0/90*
3	90/90	15/90*	0/90*	24/90*
4	0/90*	0/90*	0/90*	0/90*
5	0/90*	0/90*	0/90*	0/90*
6	0/90*	0/90*	0/90*	0/90*
7	4/90*	1/90*	0/90*	1/90*
8	0/90*	0/90*	0/90*	0/90*
9	0/90*	0/90*	0/90*	0/90*
10	0/90*	0/90*	0/90*	0/90*
11	90/90	0/90*	0/90*	0/90*
12	0/90*	0/90*	0/90*	0/90*
13	1/90*	2/90*	0/90*	3/90*
14	13/13	2/2	–	1/1
15	13/13	2/2	–	1/1

FIGURE 21: Tableau de résultats des tests du NIST pour divers exemples de générateurs avec 1 seule itération

La figure 22 donne les résultats des tests du NIST pour des générateurs faibles, mais dont le comportement se rapproche de certains PRNG implémentés dans des tags RFID connus. C'est-à-dire qu'ils ne produisent pas un nombre à chaque étape du générateur, mais après entre 10000 et 20000 étapes. Les différents LFSR sont de période maximale et de polynôme :

- (1) $x^{16} + x^{14} + x^{13} + x^{11} + 1$
- (2) $x^{32} + x^{22} + x^2 + x^1 + 1$
- (3) $x^{64} + x^{63} + x^{61} + x^{60} + 1$
- (4) $x^{16} + x^{14} + x^{13} + x^{11} + 1$ étendu sur 32 bits.
- (5) $x^{32} + x^{22} + x^2 + x^1 + 1$ étendu sur 64 bits.

Le contraste est assez flagrant vis-à-vis du tableau de la figure 21. Il est clair que les générateurs obtiennent un certain aléa statistique grâce au lecteur qui ne capte un nombre

qu'avec une précision faible.

Test	LFSR 16 bits (1)	LFSR 32 bits (2)	LFSR 64 bits (3)	LFSR 16 bits étendu sur 32 bits (4)	LFSR 32 bits étendu sur 64 bits (5)
1	90/90	89/90	90/90	89/90	89/90
2	90/90	89/90	89/90	89/90	89/90
3	89/90	89/90	87/90	90/90	88/90
4	89/90	90/90	90/90	77/90*	90/90
5	90/90	89/90	90/90	0/90*	88/90
6	87/90	89/90	88/90	90/90	89/90
7	89/90	89/90	89/90	89/90	87/90
8	89/90	90/90	89/90	88/90	89/90
9	86/90	90/90	90/90	90/90	90/90
10	89/90	90/90	90/90	90/90	88/90
11	89/90	90/90	90/90	88/90	89/90
12	89/90	90/90	87/90	88/90	87/90
13	89/90	89/90	90/90	90/90	90/90
14	50/51	53/53	55/56	61/61	60/60
15	50/51	52/53	56/56	60/61	59/60

FIGURE 22: Tableau de résultats des tests du NIST pour divers exemples de générateurs avec plusieurs itérations

Enfin, la figure 23 donne les résultats des tests du NIST pour des générateurs plus complexes, avec ou sans génération d'un nombre à chaque itération. Il faut remarquer qu'ici encore, si les tests du NIST permettent d'identifier une faiblesse dans un générateur produisant un nombre à chaque itération du générateur, ceux-ci sont incapables de détecter cette même faiblesse dès que le générateur produit plusieurs itérations avant de renvoyer un nombre.

Test	LCG Knuth & Lewis	Combinaison de LFSR de Geffe	LCG Knuth&Lewis avec plusieurs itérations	Combinaison LFSR Geffe avec plusieurs itérations
1	90/90	90/90	90/90	90/90
2	90/90	90/90	90/90	90/90
3	90/90	90/90	89/90	90/90
4	90/90	90/90	88/90	90/90
5	89/90	89/90	90/90	90/90
6	0/90*	0/90*	90/90	90/90
7	90/90	88/90	90/90	89/90
8	89/90	90/90	89/90	88/90
9	89/90	88/90	89/90	90/90
10	89/90	0/90*	90/90	90/90
11	88/90	0/90*	88/90	89/90
12	90/90	90/90	90/90	87/90
13	90/90	90/90	89/90	90/90
14	64/65	74/75	64/65	75/75
15	63/65	74/75	65/65	74/75

FIGURE 23: Tableau de résultats des tests du NIST pour divers exemples de générateurs plus complexes

8.5.2 Algorithme

Une explication des algorithmes de tests peut être trouvée dans [RSN+10].

8.5.3 Pratique

- Lancer le binaire *assess* présent dans le dossier des tests du NIST avec la commande : `./assess 1000000`
- Entrer 0 pour pour indiquer au programme qu'il doit effectuer ses tests sur un fichier de bits.
- Spécifier ensuite le chemin d'accès au fichier.
- Entrer 1 pour effectuer tous les tests du NIST sur le fichier.
- Régler les paramètres du programme (voir ci-dessous).
- Préciser au programme sur combien de séquences doit s'appliquer le test (90).
- Les résultats se trouvent dans le fichier *experiments/AlgorithmTesting/finalAnalysisReport.txt*.
- Les tests non réussis sont marqués d'une étoile (*) dans le fichier de résultats.

Paramètres

Différentes contraintes doivent être appliquées aux tests afin que ceux-ci soient valides. Ces contraintes dépendent principalement du nombre de bits de la séquence. La figure 24

présente les différentes contraintes à appliquer lors du choix des paramètres des tests du NIST. Les notations sont les suivantes :

- Le nombre de bits de la séquence est noté n.
- La taille d'un bloc (pour les tests n°2, 10, 11 et 12) est notée M.
- Le nombre de ces blocs est noté N.
- La taille d'un template (pour les tests n°7 et 8) est notée m.

N° Test	n minimum	contrainte(s)
1	100	
2	100	$n \geq NM$, $M \geq 20$, $M > 0,01n$ et $N < 100$
3	100	
4	128	
5	38912	
6	1000	
7	1000000	$m = 9$ ou $m = 10$
8	1000000	$m = 9$ ou $m = 10$
9	387840	
10	1000000	$500 \geq M \geq 5000$
11	1000000	$M < (log_2 n) - 2$
12	1000000	$M < (log_2 n) - 5$
13	100	
14	1000000	
15	1000000	

FIGURE 24: Tableau reprenant les différents paramètres des tests du NIST

Pour le fichier *nombres_non_reinit*, les paramètres de la figure 25 donnent des résultats cohérents.

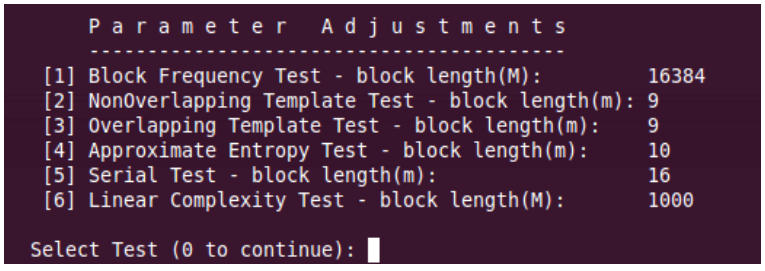

FIGURE 25: Exemple de bons paramètres à appliquer au fichier *nombres_non_reinit*

8.6 Test de dépendance des bits d'un nombre aléatoire

8.6.1 Description

Une dépendance peut éventuellement exister dans le fichier de nombres. Trouver cette dépendance peut être très intéressant car cela permet de simplifier le test final. Cela peut être une dépendance entre les nombres du fichier, ou une dépendance à l'intérieur même des nombres. Néanmoins, ce test ne s'occupe que de tester si la moitié d'un nombre aléatoire dépend de l'autre moitié. Tout simplement parce qu'il est fort peu probable qu'une autre dépendance ait lieu à l'intérieur du nombre et car cela simplifie énormément le calcul.

8.6.2 Algorithme

Le test parcourt une fois le fichier de nombres aléatoires en plaçant chaque nombre aléatoire dans une liste. Si la première moitié du nombre est déjà présente dans la liste, le test regarde la seconde moitié du nombre. Si les 2 sont égales, il passe au nombre aléatoire suivant dans le fichier. Sinon, il a trouvé deux nombres dont les premières moitiés sont égales, mais pas les secondes. La conclusion est alors qu'il n'existe pas de fonction déterministe permettant, à partir des 16 bits de poids faible du nombre, de produire les 16 bits de poids fort. Si le test ne trouve aucun nombre aléatoire violant la dépendance, c'est que celle-ci est présente, ou que l'échantillon n'est pas assez large.

8.6.3 Pratique

Lancer le programme PRNGAnalyzer.jar.
Spécifier le chemin d'accès au fichier *nombres_non_reinit*.
Effectuer le test de dépendances.
Le résultat se présente dans le fichier result.txt sous la forme :

```
Pas de dépendance entre les bits de poids faible et de poids fort
Fin du test des dépendances.
```

ou :

```
Soit le test a identifié une dépendance entre les bits de poids faible
et les bits de poids fort, soit le fichier en input est trop petit.
Fin du test des dépendances.
```

Suivant le résultat du test.

8.7 Tests de comparaison de générateurs

8.7.1 Description

Ce test consiste à comparer le PRNG analysé avec des PRNG connus, grâce aux informations recueillies dans les tests précédents. Par exemple, si le test précédent a identifié une

dépendance entre les bits de poids faible et de poids fort du nombre aléatoire, le test des générateurs ne sera effectué que sur les bits de poids fort des nombres aléatoires.

Le test va simuler des générateurs. Il va en fait prendre la liste de nombres aléatoires et regarder si, effectivement, il aurait pu générer cette liste avec le générateur testé. S'il avait pu la générer, le générateur est sans doute le bon. Sinon, le générateur est écarté. Comme le générateur du tag RFID fonctionne à priori en continu et que le lecteur a une vitesse plus faible pour récupérer les nombres aléatoires, il faudra effectuer un certain nombre d'itérations avec le générateur entre chaque nombre aléatoire. Ce test permet de trouver :

- Les LFSR à 2, 4 et 6 taps, éventuellement avec une permutation des octets du nombre aléatoire. Le test est réalisé sur toutes les permutations des taps possibles. Éventuellement, le test peut être effectué sur des générateurs plus petits que le nombre de bits du nombre aléatoire (test de tous les LFSR 16 bits sur des nombres pseudo-aléatoires de 32 bits, par exemple). Pour des nombres aléatoires de 32 bits, le test analysera 31 LFSR de 2 taps, 4495 LFSR de 4 taps et 169911 LFSR de 6 taps. S'il doit également analyser les LFSR plus petits que 32, ces nombres grimpent à 496 LFSR 2 taps, 35960 LFSR 4 taps et 906192 LFSR de 6 taps.
- Les générateurs congruentiels linéaires les plus courants (des implémentations connues et déjà existantes). Les LCG testés sont donnés dans le tableau de la figure 26. Un test exhaustif pour un modulo (par exemple : $2^{nombrebitsdunombre}$) ou des intervalles de modulo et des intervalles de multiplicateur donnés (avec ou sans somme) peut également être effectué. Cela allonge bien évidemment considérablement la durée du test.
- Les générateurs Blum-Blum-Shub, pour un intervalle de modulos donné.

Nom ou utilisation	multiplicateur a	incrément c	modulo m
RANDU	65539		2^{31}
Robert Sedgewick	31415821	1	10^8
Standard minimal	16807		$2^{32}-1$
Turbo Pascal	129	907633385	2^{32}
UNIX	1103515245	12345	2^{31}
Knuth	137	187	2^8
Knuth & Lewis	1664525	1013904223	2^{32}
Marsaglia	69069		2^{32}
Lavaux	31167285		2^{48}
Haynes	6364136223846793005		2^{64}
Borland	22695477		2^{64}
glibc	1103515245	12345	2^{32}
Haynes	134775813	1	2^{32}
Borland Delphi	6364136223846793005		2^{64}
Microsoft Visual	214013	2531011	2^{32}
Visual Basic	1140671485	12820163	2^{24}
RtlUniform	2147483629	2147483587	$2^{31}-1$
Apple CarbonLib	16807		$2^{31}-1$
MMIX	6364136223846793005	1442695040888963407	2^{64}
Java Rand	25214903917	11	2^{48}
LC35	$2^{32}-333333333$		$2^{32}-5$

FIGURE 26: Tableau des paramètres des générateurs insérés dans le test des LCG

Toute la difficulté d'un tel test est qu'il existe une infinité de générateurs, et que le choix des paramètres est important. Pour des tests plus importants, le test peut prendre énormément de temps (plusieurs heures pour un test de tous les LFSR de 64 bits et plus petits, par exemple).

8.7.2 Algorithme

Le test prend en entrée une liste de X nombres aléatoires.

L'implémentation du test des LFSR est faite de cette manière :
Le test prend en compte les LFSR dans les 2 sens (en allant du bit de poids faible jusqu'au bit de poids fort et vice-versa). Pour ce faire, il crée deux listes contenant tous les taps possibles (exemple : 16,15,14,13). Pour chaque paire de nombres aléatoires, il teste tous les taps de la première liste dans un sens, et tous les taps de la seconde liste dans le second sens. Il part donc du premier nombre aléatoire et effectue pour chaque tap un certain nombre d'itérations. Si, au cours de l'une de ces itérations, il tombe sur le second nombre aléatoire, le tap en question

est conservé et le test passe au tap suivant. Sinon, le test retire de sa liste le tap en question. Il recommence alors en prenant la seconde paire de nombres (nombres aléatoires 2 et 3), et effectue le test pour les différents taps restants dans les 2 listes.

Pour le test des générateurs linéaires congruentiels, le test procède de la même manière. Il n'y a cependant plus qu'une liste, et celle-ci contient 20 générateurs couramment utilisés en informatique. Celle-ci peut également être peuplée grâce aux paramètres passés par l'utilisateur (modulo, multiplicateur et éventuellement somme).
Le test des générateurs Blum-Blum-Shub se fait exactement de la même manière, mis à part que l'utilisateur ne peut que spécifier un intervalle de modulos.

8.7.3 Pratique

Lancer le programme PRNGAnalyzer.jar. Spécifier le chemin d'accès au fichier *nombres_non_reinit* ainsi que le nombre de nombres pseudo-aléatoires à tester. Préciser au programme le test à effectuer (LFSR, LCG, BBS). Donner au test les différents paramètres à utiliser :

- Nombre d'itérations produites par le générateur pour retrouver le second nombre aléatoire à partir du premier.
- Pour un LFSR :
  - Permutation des octets.
  - Test des LFSR plus petits.
- Pour un LCG :
  - Intervalle de modulos.
  - Intervalle de multiplicateurs.
  - Somme (Facultatif).
- Pour un BBS :
  - Intervalle de modulos.

A la fin de chaque test, un résultat est présenté, contenant :

- Le nombre de générateurs encore possibles après chaque étape.
- Pour la dernière étape (le dernier nombre aléatoire), le cas échéant, les générateurs qui auraient pu générer cette suite de nombres aléatoires pour les paramètres donnés.

9 Résultats et commentaires

Cette section présente les résultats de l'application de la procédure à différents tags.

9.1 Mifare Classic

9.1.1 Génération de nonces

Le PRNG de la Mifare Classic renvoie des nombres pseudo-aléatoires de 32 bits. Comme il faut au minimum 90 millions de bits afin d'obtenir des résultats corrects, le premier fichier généré contient 3 millions de nombres pseudo-aléatoires. En moyenne, le lecteur a pris 0,048154 seconde pour générer 1 nombre. Il aura fallu une quarantaine d'heures pour générer l'ensemble du fichier avec un lecteur SCL3711. La génération via un lecteur ACR122U est bien sûr possible, mais ce lecteur a malheureusement tendance à être plus lent lors de la génération d'aléatoire en continu.
Le second fichier est lui composé de 5000 nombres pseudo-aléatoires, dont la génération a été entrecoupée d'une réinitialisation du champ du lecteur. La procédure a été appliquée à 3 types de tags Mifare Classic différents :

- Le premier tag est un tag Mifare Classic NXP 1k standard, de bloc b0 : f3 f5 d9 cf 10 88 04 00 46 ba 14 14 55 60 07 10.
- Le second tag est un tag Mifare Classic 1k utilisé à l'UCL pour le contrôle d'accès au bâtiment. Ce tag est plus récent que le premier.
- Le dernier tag est un tag Mifare Classic Chinois, vraisemblablement sorti de l'usinage avant sa phase de finalisation. Il possède les particularités d'avoir un générateur produisant des nombres fixes et d'avoir un UID modifiable.

9.1.2 Cycle

Le programme a trouvé un cycle. Le plus petit cycle probable a été trouvé en 13 nombres. Il faut maintenant tenter de donner une approximation de la période du générateur. Étant donné que le lecteur génère un nombre en 0,048154 seconde, il produira un cycle au bout de $0,048154 * 13 = 0,626002$ seconde. Connaissant la fréquence du PRNG du tag (106 kHz), on peut estimer la période du tag à $106000 * 0,62 = 66356$ nombres. Le test visant à donner une évaluation au cycle du générateur nous donne : *Période trouvée : 65557*. Étant donné que les deux estimations sont concordantes, si le générateur ne produit pas deux nombres semblables au sein du même cycle, il est raisonnable de penser que la période de celui-ci se situe aux alentours de 66356 et 65557. Une idée intuitive au vu du fonctionnement de certains générateurs (et de leur période) serait d'estimer celle-ci à 2^{16}.

9.1.3 Graine

Le test de la graine sera effectué sur les 3 tags Mifare Classic différents.

Mifare Classic NXP 1k

Les tests basiques relatifs à la graine sont d'abord effectués sur 5000 nombres générés de la Mifare Classic NXP 1k. Les résultats de ces tests sont présentés aux figure 27 et 28.

Nombre de nombres différents	864
Nombre maximum d'occurrences d'un nombre	32

FIGURE 27: Nombre de nombres différents et maximum d'occurrences de la graine de la Mifare Classic NXP 1k

Nombre apparu	Nombre d'apparitions
01001110011110100100011000011000	32
01010111010011100111101001000110	32
00100101010001100101100111110101	26
01111010010001100001100010011101	24
11100111011010110011110101110111	21
01011001111101010000111011100001	20
01100101111001110110101100111101	19
01111100111100110101011101001110	17
01000110010110011111010100001110	16
01000110000110001001110100010101	15
10011000000111011010000101110001	15
11110011010101110100111001111010	14
00101010111011011010110000101100	13
00111100101011110110110100011100	13
10000000100011000101010010011010	13
11010010010011101100101011000110	12
10110011100000001000110001010100	11
11000100110110000010101100001011	11
11110101011101100010000100001110	11
11101101101011000010110001110111	11

FIGURE 28: Nombres d'occurrences des 20 nombres pseudo-aléatoires apparaissant le plus de fois pour la graine de la graine de la Mifare Classic NXP 1k

Il est clair qu'intuitivement, la graine ne semble pas aléatoire. En effet, le tableau de

la figure 28 présente des valeurs étonnantes d'occurrence pour 5000 nombres parmi 65535 possibilités. De plus, la figure 27, montre que seulement 864 nombres sont différents parmi les 5000 tirés. Comme la période du générateur n'est cependant pas encore connue avec certitude et précision, seule une intuition sur le caractère aléatoire de la graine peut être tirée.

A posteriori, une fois le générateur de la Mifare Classic connu, il est cependant possible de tester le caractère aléatoire de la graine avec plus de précision. La division des 5000 nombres obtenus en 100 catégories permet d'obtenir les résultats présentés dans les figures 29 et 30.

–

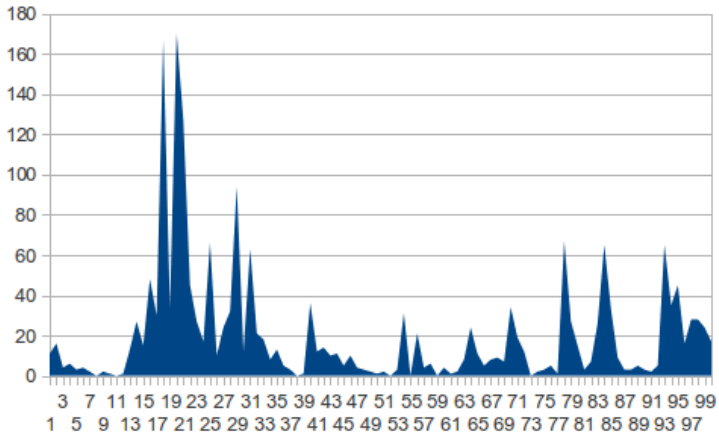

FIGURE 29: Graphique de l'intervalle des index des graines pour une Mifare Classic NXP 1k

Valeur	Interprétation
4794.799	Pas aléatoire
7952.08	Pas aléatoire
7648.08	Pas aléatoire

FIGURE 30: Résultats et interprétation du test chi-carré pour la graine de la Mifare Classic NXP 1k

Graphiquement, le résultat semble assez clair. En effet, sur le graphique de la figure 29, des pics de valeurs se forment et les différents nombres possibles ne sont pas étalés sur l'ensemble des index des possibilités. Le test du chi-carré présenté à la figure 30 vient confirmer l'intuition de départ. Le test, effectué sur trois fichiers générés indépendamment renvoie des valeurs très éloignées d'une distribution aléatoire optimale. Comme les trois tests ont échoué, il est possible de conclure que la graine n'est pas aléatoire. Étant donné les résultats, il est même possible de penser que la graine de la Mifare Classic NXP 1k est fixe et que si le lecteur était assez précis, le générateur produirait toujours le même nombre.

Mifare Classic 1k UCL

Les tests basiques relatifs à la graine sont d'abord effectués sur 5000 nombres générés de la Mifare Classic 1k UCL. Les résultats de ces tests sont présentés aux figure 31 et 32.

Nombre de nombres différents	4813
Nombre max d'occurrences d'un nombre	3

FIGURE 31: Nombre de nombres différents et maximum d'occurrences de la graine de la Mifare Classic 1k UCL

Nombre apparu	Nombre d'apparitions
10111110100010110111101111000100	3
10010110011101100101010100101110	3
00100100111101111011000001010011	3
01001011000001100010001011010111	3
10101000110011111100101001110011	2
01101111011001001101101001100010	2
11000010000100111101010000110101	2
00010110111101111000100111111011	2
11100110001100111111110001011000	2
10000001111001111010110101001101	2
10110110000011010000000000001111	2
01010100110111101111100101011100	2
01011000111000111111111100011000	2
10111001000001101111010101001111	2
00110101000111101000111101000010	2
10000100111000110000100111101000	2
00101001001010010000111101111111	2
10010100100110101011011100110011	2
00001100010111010000011000110011	2
01110100100001100001010010010011	2

FIGURE 32: Nombre d'occurrences des 20 nombres pseudo-aléatoires apparaissant le plus de fois pour la graine de la graine de la Mifare Classic 1k UCL

En comparaison avec la Mifare Classic NXP 1k, il semble que cette graine soit bien meilleure. Un nombre n'apparaît en effet que 3 fois au maximum parmi les 5000 nombres. De plus, 4813 nombres sont différents parmi les 5000 obtenus. Il est cependant impossible de conclure que cette graine est aléatoire (le nombre total de valeurs possibles est inconnu). Comme le générateur de la Mifare Classic est connu, il est cependant possible de tester le

caractère aléatoire de la graine avec plus de précision. La division des 5000 nombres obtenus en 100 catégories permet d'obtenir les résultats présentés dans les figures 33 et 34.

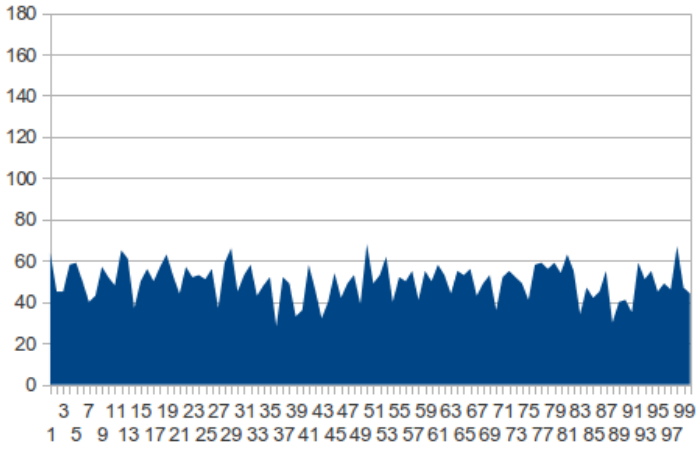

FIGURE 33: Graphique de l'intervalle des index des graines pour une Mifare Classic 1k UCL

Valeur	Interprétation
128.36	Suspect
97.38	Aléatoire
110.9	Aléatoire

FIGURE 34: Résultats et interprétation du test chi-carré pour la graine de la Mifare Classic 1k UCL

Le graphique de la figure 33 est bien différent de celui de la figure 29. Ici, il n'y a plus de pics clairs qui se forment, et les nombres sont bien plus étalés sur l'ensemble de toutes les valeurs possibles. Le test du chi-carré présenté à la figure 34 vient confirmer les impressions du graphique. Le test, effectué sur trois fichiers générés indépendamment renvoie deux fois le résultat aléatoire, et une fois un résultat suspect. Ce dernier résultat est sans doute dû à un échantillon très défavorable. Comme précisé à la section 6.1, comme deux tests sur trois sont concluants, la graine peut être dite aléatoire au sens du test Chi-carré.

Mifare Classic "Chinoise"

2 tests ont été effectués sur les 5000 nombres générés de la Mifare Classic "Chinoise". Les résultats de ces tests sont présentés aux figure 35 et 36.

Nombre de nombres différents	1
Nombre max d'occurrences d'un nombre	5000

FIGURE 35: Nombre de nombres différents et maximum d'occurrences de la graine de la Mifare Classic "chinoise"

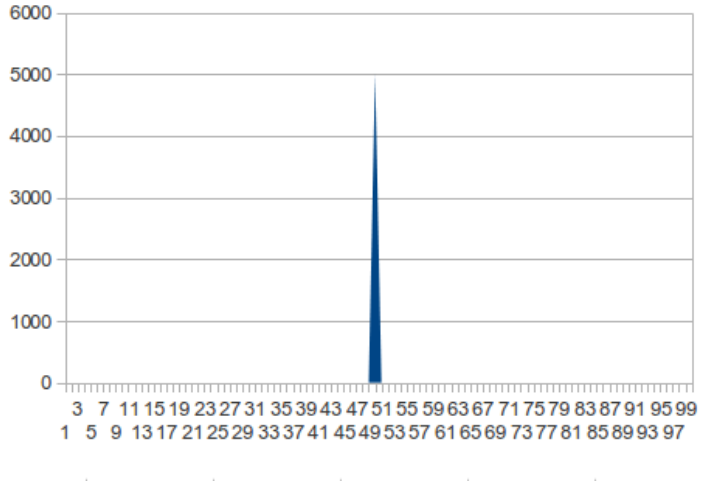

FIGURE 36: Graphique de l'intervalle des index des graines pour une Mifare Classic "chinoise"

Comme le montrent les figures 35 et 36, le "générateur" de ce type de Mifare Classic renvoie indéfiniment le même nombre. La graine (si l'on peut parler de graine), est donc évidemment fixe.

9.1.4 Tentative d'identification d'une graine fixe semblable pour plusieurs tags

Ce complément au test de la graine a pour but de comparer les différentes valeurs obtenues par le test de la graine pour une Mifare Classic NXP 1k. Comme précisé dans le test précédent, les différentes graines de ces tags sont fixes. L'idée de ce test est donc de comparer plusieurs tags afin de vérifier si la graine semble être la même sur les différents tags. Ce test a été effectué sur 3 tags différents. Chaque tag ayant été testé 3 fois. Les figures 37, 38 et 39 présentent les graphiques des index de valeurs obtenus pour 3 générations différentes de 5000 nombres avec réinitialisation du champ du lecteur effectuées sur chacun des 3 tags.

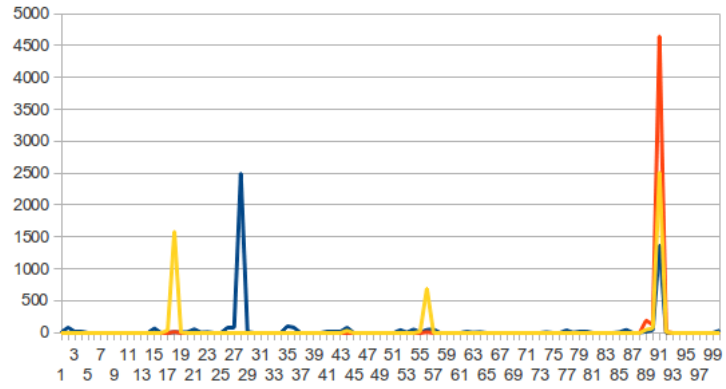

FIGURE 37: Graphique de l'intervalle des index des graines pour la Mifare Classic NXP 1k n°1

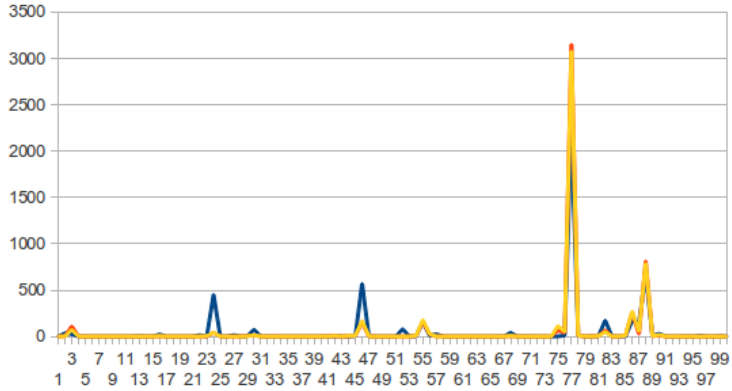

FIGURE 38: Graphique de l'intervalle des index des graines pour la Mifare Classic NXP 1k n°2

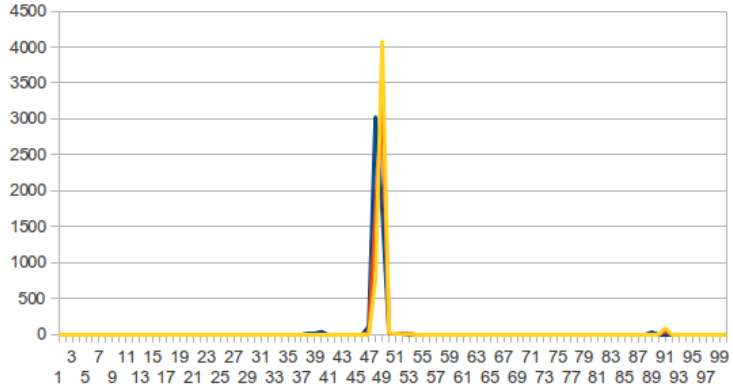

FIGURE 39: Graphique de l'intervalle des index des graines pour la Mifare Classic NXP 1k n°3

Pour les générations de chacun des 3 tags, des pics se retrouvent aux mêmes endroits. Les différentes courbes des différents graphiques semblent concorder et se superposer sur chacun des graphes. Cela signifie qu'il n'y a pas eu de perturbations lors des générations, et que les 3 générations successibles semblent indiquer que le tag possède la même graine.

Un tel graphe présenté pour des Mifare Classic différentes mais possédant la même graine fixe, devrait donc présenter le même genre de courbes superposées qu'obtenues dans les figures 37, 38 et 39. Le graphique de la figure 40 présente une telle expérience.

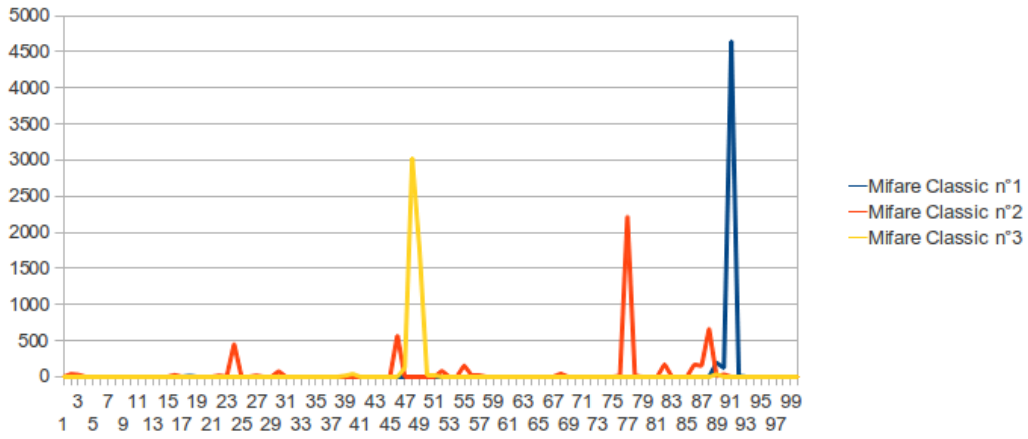

FIGURE 40: Graphique de l'intervalle des index des graines des 3 Mifare Classic NXP 1k

Le graphique de la figure 40 donne un résultat totalement différent. En effet, ce graphe ne présente pas des courbes se superposant. Les pics n'interviennent pas pour les mêmes valeurs d'index. Plus précisément, les plus hauts pics constatés se retrouvent dans la catégorie 91

pour la Mifare Classic n°1, la catégorie 77 pour la Mifare Classic n°2 et la catégorie 48 pour la Mifare Classic n°1. Il est donc impossible de conclure via cette expérience que les graines de Mifare Classic différentes sont les mêmes. Au contraire, cette expérience soulève plusieurs questions :

- Les graines fixes de Mifare différentes sont-elles différentes ? Est-il dès lors possible d'identifier un tag Mifare Classic en fonction de sa graine ?
- Ces valeurs sont-elles les mêmes lorsque l'expérience est réalisée dans d'autres conditions ? En modifiant la position du tag par rapport au lecteur, par exemple.
- Est-ce que certains tags prennent plus de temps que d'autres à renvoyer un nombre ? En particulier, le temps pris par le tag pour effectuer le protocole d'anti-collision pourrait être différent d'un tag à l'autre.

L'expérience a donc été reproduite une seconde fois afin de pouvoir préciser nos résultats. Le résultat de la seconde expérimentation est présenté sur la figure 41

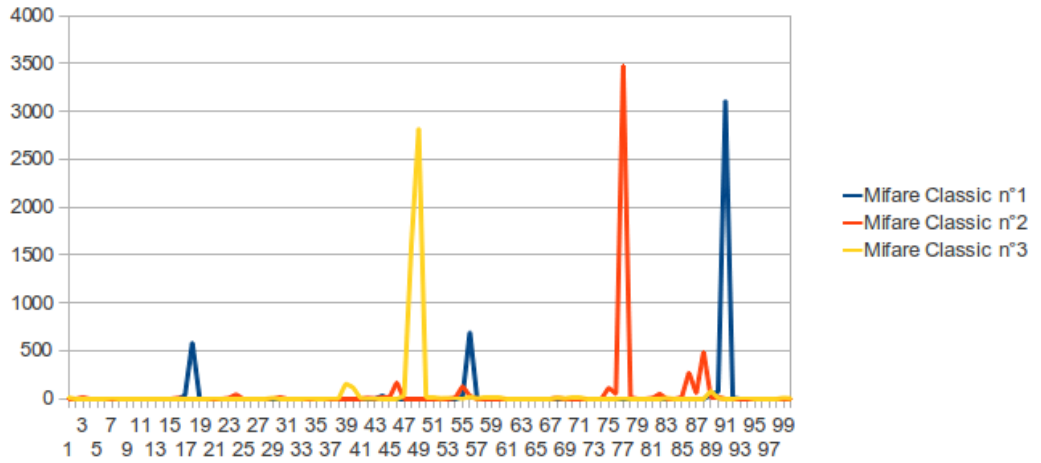

FIGURE 41: Graphique de l'intervalle des index des graines des 3 Mifare Classic NXP 1k pour la seconde expérience

Le graphique de la seconde expérience, réalisée à un autre moment et donc en ayant déplacé le tag vis-à-vis du lecteur est pratiquement le même que le graphique de la figure 40. En particulier les, plus hauts pics constatés se retrouvent dans les mêmes catégories que lors du test précédent. Ce résultat vient donc éliminer la seconde hypothèse faite sur les graines fixes de Mifare Classic. Il ne reste donc plus que deux possibilités : Soit les graines fixes de Mifare Classic différentes sont différentes, soit certains protocoles exécutés avant l'envoi de nombre par le tag prennent plus ou moins de temps, en fonction du tag.

9.1.5 Tests statistiques

Le PRNG de la Mifare Classic échoue à 2 tests du NIST, comme le montre la figure 42. Le premier, le rank test, n'est d'ailleurs passé avec succès par aucune des 90 séquences d'un million de bits. Cet échec au rank test suppose la présence d'une dépendance entre les nombres, ou d'un PRNG sous la forme d'un LFSR. Le second test échoué, le serial test, indique que certains patterns se retrouvent plus souvent que d'autres dans les différentes séquences générées.

Test	Pass Rate
Fréquence	88/90
Fréquence dans un bloc	89/90
Runs	89/90
Plus long run de 1	86/90
Rank	0/90 *
Fourier	90/90
Non-overlapping Template Matchings	89/90
overlapping Template Matchings	88/90
Universal	88/90
Linear Complexity	89/90
Serial	48/90 *
Entropie approximative	89/90
Sommes cumulatives	88/90
Random Excursions	56/57
Random Excursion Variant	55/57

FIGURE 42: Tableau de résultats des tests du NIST pour une Mifare Classic

9.1.6 Dépendance

Le test renvoie :

Soit le test a identifié une dépendance entre les bits de poids faible et les bits de poids fort, soit le fichier en input est trop petit.

Il est fort improbable que le fichier soit trop petit. En effet, si il existe une dépendance entre les bits de poids fort et les bits de poids faible du nombre, il n'existe que $2^{16} = 65536$ possibilités de nombres différents au maximum, le reste du fichier de 3 millions de nombres ne contiendrait plus que des doubles. Or, s'il n'y a pas de dépendance, il est fort probable que le test identifie un nombre dont les premières moitiés sont semblables et les secondes moitiés

différentes. Le test n'en a identifié aucun, il est donc pertinent de suspecter la présence d'une dépendance entre les bits de poids fort et les bits de poids faible des nombres.

9.1.7 Générateur

Les tests précédents ont donné une estimation de la période à 66356 nombres. C'est une période maximale, et le générateur a sans doute une période plus courte. Or, $2^{16} = 65536$, il est donc raisonnable de commencer les tests sur des générateurs de période maximale 2^{16}. Un LCG de période maximale de forme

$$X_{n+1} = (aX_n + c) \bmod 2^{16},\ n \geq 0$$

ou un LFSR de 16 bits de période maximale semblent parfaitement adaptés pour produire une telle séquence.

Le premier test, effectué sur des LCG de modulo 2^{16}, $2^{16}+1$ et $2^{16}-1$ pour 10 nombres et 20 000 itérations du LCG ne donnera aucun résultat.

Le test effectué sur les LFSR obtiendra plusieurs résultats, pour 10 nombres et 20000 itérations du générateur. Le test sera également effectué sur les séquences dont les octets ont été inversés. Les LFSR suivants ont été identifiés par le test avec une inversion des octets :

- $x^{16} + x^{14} + x^{13} + x^{11} + 1$
- $x^{32} + x^{20} + x^{12} + x^{9} + 1$
- $x^{32} + x^{28} + x^{26} + x^{22} + 1$

Il va de soi que les différents LFSR identifiés produisent en fait la même séquence (les taps du 3^{eme} LFSR trouvé sont le double des taps du premier LFSR identifié). Il est donc possible de limiter le résultat à un seul de ces générateurs :

$$x^{16} + x^{14} + x^{13} + x^{11} + 1$$

Afin de valider ce résultat, le test peut être effectué sur un plus grand nombre de nombres, et donne le même résultat. Il est donc possible de penser que le générateur de la Mifare Classic a été identifié grâce à l'application de la procédure.

9.1.8 Conclusion

La procédure, appliquée à un tag de type Mifare Classic permet de trouver un générateur probable pour la Mifare Classic, un LFSR, dont le polynome est :

$$x^{16} + x^{14} + x^{13} + x^{11} + 1$$

La procédure a également permis de trouver plusieurs faiblesses :

- La période maximale courte du générateur, d'environ 66356 nombres.
- Une dépendance probable entre les bits de poids faible et de poids fort des nombres générés.
- Une faiblesse de la graine de certains types de Mifare Classic, qui semble être fixe.

9.2 Mifare DESFire MF3IC41-EV1

9.2.1 Génération de nonces

Le PRNG de la Mifare DESFire renvoie des nombres pseudo-aléatoires de 64 bits. Comme il faut au minimum 90 millions de bits afin d'obtenir des résultats corrects, le premier fichier généré contient 1,5 millions de nombres pseudo-aléatoires.
Le second fichier est lui composé de 5000 nombres pseudo-aléatoires, dont la génération a été entrecoupée d'une réinitialisation du champ du lecteur.

9.2.2 Cycle

Le programme n'a pas pu trouver de double parmi les 1,5 millions de nombres pseudo-aléatoires. Le générateur a vraisemblablement une période plus élevée que 1500000.

9.2.3 Graine

Pour la Mifare DESFire, le test de la graine sur le fichier de 5000 nombres aléatoires donne comme résultat 5000 nombres différents. Les 5000 nombres n'apparaissent donc chacun qu'une seule fois dans le fichier. On peut en conclure que la graine n'est pas fixe.

9.2.4 Tests statistiques

Test	Pass Rate
Fréquence	90/90
Fréquence dans un bloc	90/90
Runs	90/90
Plus long run de 1	90/90
Rank	86/90
Fourier	86/90
Non-overlapping Template Matchings	88/90
overlapping Template Matchings	90/90
Universal	86/90
Linear Complexity	90/90
Serial	90/90
Entropie approximative	90/90
Sommes cumulatives	90/90
Random Excursions	55/57
Random Excursion Variant	56/57

FIGURE 43: Tableau de résultats des tests du NIST pour une Mifare DESFire

Le fichier de 90 millions de bits de la Mifare DESFire passe avec succès tous les tests statistiques du NIST. La figure 43 présente ces résultats.

9.2.5 Dépendance

Aucune paire de nombres violant la dépendance n'a été trouvée (une moitié de bits semblable avec l'autre moitié différente). Le fichier d'input est cependant trop petit (par rapport à la période probable) pour conclure à l'existence une dépendance.

9.2.6 Générateur

Aucun LFSR de 2, 4 ou 6 taps ne correspond avec les nombres pseudo-aléatoires générés par la Mifare DESFire, et ce avec ou sans inversion des octets du nombre. Aucun LCG connu ne correspond à la suite de nombres aléatoires générée par la Mifare DESFire. Aucun test supplémentaire sur les LCG n'a été effectué étant donné l'absence de résultat des tests précédents.

9.2.7 Conclusion

La procédure appliquée au PRNG de la Mifare DESFire n'a permis d'identifier aucune faiblesse. Les résultats semblent indiquer que le générateur implémenté dans ce tag produit de bons nombres pseudo-aléatoires. La période du PRNG de la DESFire semble supérieure à 1,5 million. De plus, la graine de ce générateur n'est vraisemblablement pas fixe.

9.3 Mifare Ultralight C

9.3.1 Génération de nonces

Le PRNG de la Mifare Ultralight C, tout comme celui de la Mifare DESFire, renvoie des nombres pseudo-aléatoires de 64 bits. Comme il faut au minimum 90 millions de bits afin d'obtenir des résultats corrects, le premier fichier généré contient 1,5 millions de nombres pseudo-aléatoires.
Le second fichier est lui composé de 5000 nombres pseudo-aléatoires, dont la génération a été entrecoupée d'une réinitialisation du champ du lecteur.

9.3.2 Cycle

Le programme n'a pas pu trouver de double parmi les 1,5 millions de nombres pseudo-aléatoires. Le générateur a vraisemblablement une période plus élevée que 1500000.

9.3.3 Graine

Pour la Mifare Ultralight C, le test de la graine sur le fichier de 5000 nombres aléatoires donne comme résultat 5000 nombres différents. Les 5000 nombres n'apparaissent donc chacun qu'une seule fois dans le fichier. On peut en conclure que la graine n'est pas fixe.

9.3.4 Tests statistiques

Test	Pass Rate
Fréquence	87/90
Fréquence dans un bloc	90/90
Runs	90/90
Plus long run de 1	89/90
Rank	90/90
Fourier	88/90
Non-overlapping Template Matchings	89/90
overlapping Template Matchings	88/90
Universal	88/90
Linear Complexity	90/90
Serial	90/90
Entropie approximative	89/90
Sommes cumulatives	87/90
Random Excursions	57/57
Random Excursion Variant	56/57

FIGURE 44: Tableau de résultats des tests du NIST pour une Mifare Ultralight C

Le fichier de 90 millions de bits de la Mifare Ultralight C passe avec succès tous les tests statistiques du NIST. La figure 44 présente ces résultats.

9.3.5 Dépendance

Le test a identifié une absence de dépendance (une moitié de bits semblable avec l'autre moitié différente).

9.3.6 Générateur

Aucun LFSR de 2, 4 ou 6 taps ne correspond avec les nombres pseudo-aléatoires générés par la Mifare Ultralight C, et ce avec ou sans inversion des octets du nombre. Aucun LCG connu ne correspond à la suite de nombres aléatoires générée par la Mifare Ultralight C. Aucun test supplémentaire sur les LCG n'a été effectué étant donné l'absence de résultat des tests précédents.

9.3.7 Conclusion

La procédure appliquée au PRNG de la Mifare Ultralight C n'a permis d'identifier aucune faiblesse. Les résultats semblent indiquer que le générateur implémenté dans ce tag produit de bons nombres pseudo-aléatoires. La période du PRNG de la Mifare Ultralight C est sans

doute supérieure à 1,5 million. De plus, la graine de ce générateur n'est vraisemblablement pas fixe.

9.4 Commentaires

Après application de la procédure aux quelques exemples réels présentés ci-dessus, il convient de donner quelques commentaires à chaque étape de la procédure.

9.4.1 Génération de nombres aléatoires

Sans être une étape difficile, plusieurs problèmes peuvent se poser. Premièrement, certains tags effectuent l'étape d'authentification à plus bas niveau que les APDU. C'est notamment le cas de la Mifare Classic. Il est alors nécessaire de trouver une librairie adaptée permettant de communiquer avec le tag à plus bas niveau afin de pouvoir enregistrer les nombres aléatoires produits par celui-ci. Le problème s'est également posé avec les tags respectant l'ISO 15693. L'utilisation d'une librairie telle que librfid est de trop haut niveau pour pouvoir capter les nombres aléatoires produits par le tag. En second lieu, la génération de nombres aléatoires est assez longue étant donnée la quantité de nombres nécessaires pour effectuer les tests. De plus, celle-ci doit se dérouler en une seule fois afin de ne pas compromettre les résultats.

9.4.2 Cycle

Il est raisonnable de penser qu'un tel test ne pourra identifier un cycle que pour des générateurs très faibles. Le temps nécessaire pour la génération d'une grande quantité de nombres est en effet très élevé (plusieurs jours pour seulement 3 millions de nombres). Or, un simple LFSR à 32 bits de période maximale produit $2^{32} - 1 = 4294967296$ itérations avant de produire un cycle. Porté à la fréquence d'une Mifare Classic par exemple, cela signifie qu'il faudra plus de 11 heures à un tel générateur avant de produire un cycle (contre moins d'une seconde pour un LFSR 16 bits). Il faudra même plus de 5 millions d'années à un LFSR 64 bits de même fréquence pour produire un cycle. Et comme le lecteur ne renvoie qu'un nombre sur quelques milliers d'itérations du PRNG à chaque requête d'un nombre, il est possible de penser que ce test ne pourra jamais identifier de générateur dont la période est suffisamment grande. C'est vraisemblablement pour cette raison que les Mifare DESFire et Ultralight C ne renvoient aucun double pour ce test.

9.4.3 Graine

Le test de la graine s'intéresse à la base du PRNG et est donc très intéressant. En effet, avec matériel relativement basique (un simple SCL3711), il a été montré que dans le cas d'une graine fixe, il était possible de produire 32 fois le même nombre pour 5000 générations. Quelque soit le générateur implémenté, celui-ci est directement compromis s'il utilise une graine fixe. Néanmoins, ce test ne nous apprend absolument rien sur le PRNG implémenté dans le tag analysé.

Une analyse statistique pourrait être très intéressante sur un fichier de nombres générés avec réactivation du champ du lecteur entre chaque génération. En effet, une bonne graine doit être réellement aléatoire et donc générée par des moyens physiques. Malheureusement, il est impossible de remonter aux graines sans connaître le générateur utilisé et le nombre d'itérations produite par le générateur depuis son initialisation pour produire un nombre.

9.4.4 Tests statistiques

Une simple analyse des tableaux de résultats de plusieurs générateurs présentés à la section 8.5.1 permet de dire que :

- La génération de nonces en continu et l'imprécision du lecteur rendent l'identification de faiblesses très difficile via des tests statistiques. Même des générateurs très simples (un LFSR 16 bits par exemple) passent tous les tests du NIST moyennant cette caractéristique.
- Le seul générateur continu analysé ne réussissant pas tous les tests avec succès est celui de la Mifare classic (LFSR 16 bits étendu sur 32).
- Des générateurs plus compliqués ne générant des nombres que lors des interrogations du lecteur échouent à différents tests.

Dès lors, si les tests statistiques ne sont pas tous passés avec succès, il est possible de conclure que le générateur analysé :

- Soit souffre d'une réelle erreur d'implémentation (générateur 16 bits étendu sur 32 et provoquant des dépendances, par exemple).
- Soit ne génère des nonces qu'à chaque interrogation du lecteur.

Il est également évident que le fait de réussir tous les tests du NIST pour un générateur n'en fait pas un bon générateur.

9.4.5 Dépendance

Le test des dépendances est très spécifique et n'a été implémenté que parce que le générateur de la Mifare Classic présentait cette faiblesse. Néanmoins, si ce genre de faiblesse est présent dans un tag comme la Mifare Classic, il est fort possible que la même faiblesse se retrouve dans un autre tag.

9.4.6 Générateur

Bien que ce test permette d'identifier le générateur utilisé dans un tag, il présente certaines limites. Premièrement, certains types de PRNG implémentés dans la technologie ne sont pas identifiables via ce test (LFG, fonction de filtrage). D'autres types de PRNG ne peuvent pas être trouvés dans des temps raisonnables. Il est par exemple impossible de tester toutes les combinaisons de LFSR pouvant exister. Néanmoins, pour des LFSR simples, cette méthode donne d'excellents résultats.
Il est également difficilement concevable qu'un tag RFID implémente un générateur Blum-Blum-Shub, par exemple. A priori, les générateurs de type LFSR (simple, combinaison, filtré)

seraient les plus présents dans la technologie RFID pour leur facilité d'implémentation (de simples portes logiques) et leur rapidité.

Quatrième partie

Événements extérieurs pouvant influencer le PRNG d'un tag RFID

10 Tests et objectifs

Cette partie décrit les expériences qui ont été réalisées sur un tag RFID. Les tags RFID présentent des limites de fonctionnement, notamment des limites de températures normales de fonctionnement. Des tests ont été mis en place pour chaque expérience, qui permettent de vérifier que le générateur pseudo-aléatoire du tag fonctionne toujours normalement, c'est à dire de la même façon que dans un environnement normal de fonctionnement. Ces tests seront effectués sur un tag Mifare Classic. Comme le PRNG de ce tag est connu, la détection d'une erreur dans le fonctionnement de son LFSR sera aisée. 3 tests seront effectués, pour contrôler respectivement les résultats, la vitesse et la graine de ce PRNG. Il est bien sûr impossible d'effectuer dans ce cas-ci un test demandant une grande génération de nombres, car les événements extérieurs appliqués au tag sont de courte durée. Aucun document présentant des expériences semblables n'a pu être trouvé dans la littérature, celles-ci ont donc été élaborées en se basant principalement sur les spécifications des limites de fonctionnement de la Mifare Classic.

10.1 Test de la validité d'un nombre

Le premier test a pour but de contrôler les résultats du PRNG. Le tag Mifare Classic implémente un LFSR de 16 bits qui produit des nombres pseudo-aléatoires de 32 bits. Il est possible, comme la section 7.3 l'a précisé, de reconstituer les 16 bits de poids faible à partir des 16 bits de poids fort. Pour ce faire, il suffit de partir des 16 bits de poids fort, et de produire les 16 bits de poids faible à partir du LFSR implémenté dans la Mifare Classic, à savoir :

$$x^{16} + x^{14} + x^{13} + x^{11} + 1$$

Pour chaque combinaison des 16 premiers bits, il n'y a qu'une et une seule valeur possible pour les 16 derniers bits. Le test aura donc pour but de vérifier si le nombre produit par le PRNG de la Mifare Classic est un nombre pouvant effectivement avoir été généré par la Mifare Classic. Pour chaque nombre produit par le tag, le test vérifiera les 16 bits de poids faible en fonction des 16 bits de poids fort. Si le test identifie un nombre dont les 16 bits de poids faible ne peuvent pas être retrouvés à partir des 16 bits de poids fort et du LFSR de la Mifare Classic, cela voudra dire que ce nombre ne pourrait pas être généré par une Mifare Classic dans des conditions normales, et que les événements extérieurs appliqués au tag ont perturbé son générateur.

10.2 Test de la distance entre 2 nombres

Le second test a pour but de contrôler la vitesse de génération du PRNG. Étant donné que le lecteur, moyennant quelques imprécisions, récupère des nombres à intervalle régulier, si le test détecte une valeur dépassant une borne fixée, le générateur aura été perturbé. Le tag produit un nombre x_1 et l'envoie. Celui-ci continue ensuite sa génération. Le prochain nombre envoyé au lecteur sera le nombre $x_{1+\delta\pm\rho}$, où δ est le nombre d'itérations effectué par le LFSR de la Mifare Classic entre 2 interrogations, et ρ l'imprécision liée au lecteur. ρ pour un tag Mifare Classic et un générateur SCL3711 dans l'environnement utilisé a été évalué à un maximum de 2000 itérations tandis que δ a été évalué à 6000. Si lors du test, le tag produit des nombres avec un écart de moins de 4000 ou de plus de 8000 itérations de son LFSR, cela voudra dire que le LFSR de la Mifare Classic génère des nombres respectivement moins ou plus vite que lors de son fonctionnement normal. L'événement extérieur appliqué influencerait alors la vitesse de génération du PRNG.

10.3 Test de la graine

Le dernier test a pour but de contrôler la graine du PRNG. Ici, le test aura simplement comme implémentation le test vu en 8.4. Il sera effectué si possible sur un tag mifare classic de graine non fixe. Si des anomalies par rapport au fonctionnement normal (tel que décrit dans la section 9.1.3) du tag apparaissent, l'événement extérieur appliqué au tag modifie la génération de la graine du PRNG. Ce test est le seul qui demande une génération préalable de nombres, et qui ne peut pas être effectué au cours de celle-ci. Néanmoins, comme le test de la section 8.4 ne demande que 5000 nombres, il est possible d'effectuer une telle génération sur la durée de la perturbation appliquée. Ce test n'a pas réellement pour but de tester le générateur en lui-même mais plutôt la façon dont est générée sa graine.

11 Evénements extérieurs

11.1 Désencapsulation d'un tag RFID

Les expériences, et plus particulièrement celles des sections 11.2 et 11.3 ne pouvaient pas directement être effectuées sur la carte Mifare Classic. En effet, le microprocesseur de ce tag bénéficie de plusieurs couches de protection rendant impossible les expériences directes sur celui-ci. Ces tentatives d'isolation ont d'abord été menées sur des Mifare Classic présentées sous la forme d'un tag autocollant, où le microprocesseur semblait plus facilement accessible. La figure 45 présente un de ces tags, où le microprocesseur entouré de ses couches de protection est la partie noire du tag. Cette protection est notamment formée par une couche de silicium, et les différentes pistes envisagées afin de l'isoler (attaque chimique via de l'acide fluorhydrique par exemple) n'ont pas pu être menées à bien.

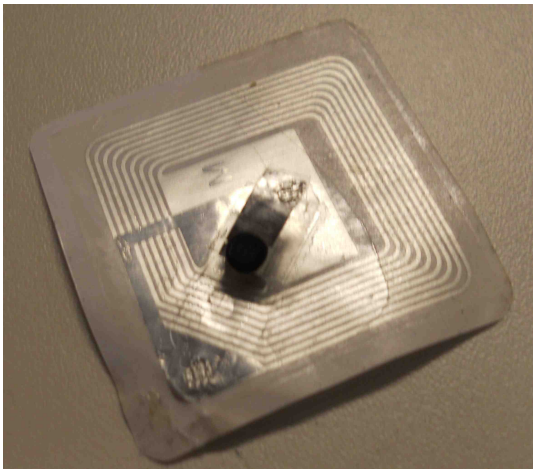

FIGURE 45: Tag Mifare Classic autocollant

La seconde technique a été mise en œuvre sur un tag Mifare Classic présenté sous la forme d'une carte plastique. L'acétone permet de dissoudre entièrement la couche de plastique entourant le tag. Néanmoins, le microprocesseur reste protégé par au moins une couche supplémentaire. La méthode d'isolation choisie a finalement été un simple fraisage du tag à l'endroit du microprocesseur. Cette méthode, bien que plus grossière, est celle qui permet de retirer le plus de couches protégeant le microprocesseur sans recourir à l'utilisation de moyens plus lourds. La figure 46 présente l'isolation réalisée sur le tag telle qu'elle a été appliquée pour réaliser les expériences.

FIGURE 46: Isolation du microprocesseur réalisée pour les expériences

11.2 Tirs de Laser

Le premier test a eu pour but d'effectuer des tirs de laser sur un tag Mifare Classic. Le laser utilisé possède une puissance de fonctionnement maximale de 500mW ainsi qu'une longueur d'onde de 700-960 nm. La figure 47 présente une photo des conditions de l'expérience. Les 2 premiers tests de la section 10 ont été appliqués sur le tag dans ces conditions. Le test de la graine n'a pas pu être réalisé sur un tag Mifare de graine non fixe, étant donné que le microprocesseur d'un tel tag n'a pas été isolé. Ces tests ont été effectués sous un tir continu du laser. Le tir du laser s'est lui concentré sur le microprocesseur du tag RFID.

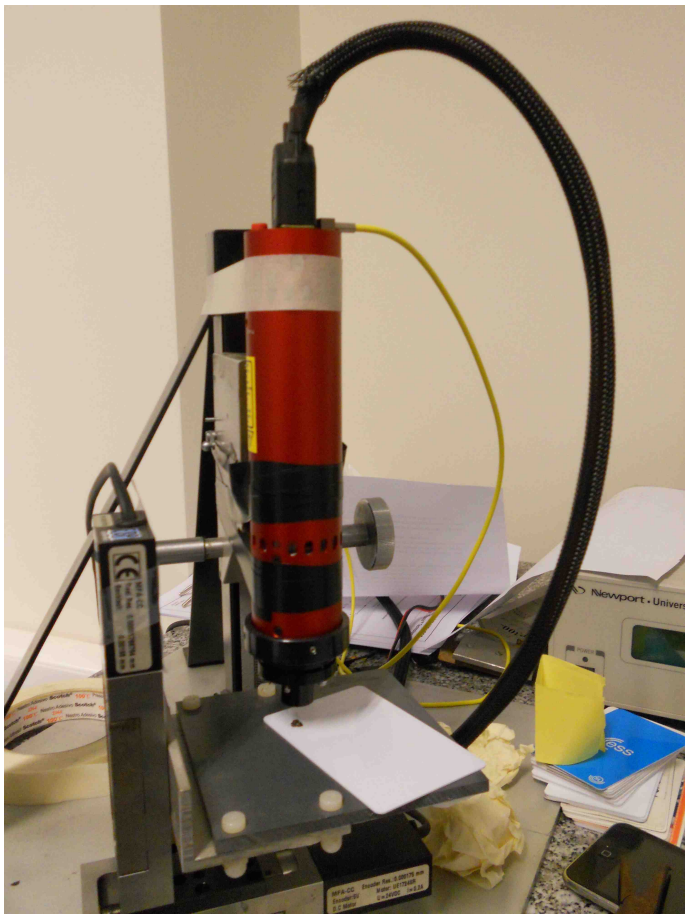

FIGURE 47: Présentation de l'expérience réalisée avec le laser

Résultats et commentaires

Tous les nombres produits par le LFSR de la Mifare Classic au cours de l'expérience ont été validés par le test et sont donc des nombres pouvant être produits par le tag dans un environnement normal. De même, tous les nombres générés l'ont été dans un intervalle d'itérations du LFSR compris entre 4000 itérations et 8000 itérations. Cela signifie que la génération de nombres du LFSR de la Mifare Classic n'a pas été altérée (ou altérée suffisamment) pour que le test de la distance entre deux nombres échoue. Bien que ce test n'a été réalisé que sur un tag présentant une graine fixe, le générateur de celui-ci a présenté des résultats équivalents à ceux obtenus à la section 9.1.3 pour une graine fixe.

11.3 Refroidissement du tag via des bombes réfrigérantes

Il est précisé dans les spécifications du tag [NXP11b] les valeurs limites de température de fonctionnement de la Mifare Classic. Celles-ci sont situées entre -55°C et 125°C en stockage, et entre -25°C et 75°C pour la température ambiante. En refroidissant le tag à cette limite, le

fonctionnement de celle-ci devrait donc être perturbé. Les spécifications précisent d'ailleurs que des variations au-delà des limites de températures précisées peuvent causer des dégâts permanents aux tags et qu'une exposition prolongée à des valeurs proches de ces limites peut affecter la fiabilité du tag. Afin d'effectuer ces tests, une bombe réfrigérante servant à refroidir certains composants électroniques afin d'identifier des pannes matérielles a été utilisée. La bombe utilisée est capable de refroidir un composant à -50˚C et -65˚C en pointe [12]. L'application de froid s'est faite sur un tag passé par le processus d'isolation décrit à la section 11.1. Les 2 premiers tests de la section 10 ont été appliqués sur le tag dans ces conditions. Le test de la graine n'a pas pu être réalisé sur un tag Mifare de graine non fixe pour les mêmes raisons que l'expérience de la section 11.2.

Résultats et commentaires

Les trois tests effectués lors de cette expérience n'ont montré aucun changement par rapport aux tests effectués dans des conditions normales de fonctionnement. Il semblerait que la Mifare Classic (ou tout du moins son PRNG) offre un fonctionnement normal dans ces conditions. En particulier, les tests n'ont renvoyé aucun nombre impossible à générer pour une Mifare Classic, la vitesse de fonctionnement est restée dans les bornes fixées par le test de la distance entre 2 nombres et le test de la graine pour une Mifare Classic a présenté des résultats cohérents pour une graine fixe. Alors que ces températures approchaient ou dépassaient les limites précisées par les spécifications, le tag n'a donc subi aucune perturbation visible par les trois tests effectués, et son PRNG n'a vraisemblablement subi aucun dysfonctionnement. Le test de la section 11.4 vise donc à refroidir à une température encore plus basse le tag RFID afin d'obtenir une perturbation.

11.4 Refroidissement du tag via de l'azote liquide

Le test de la section 11.3 n'ayant pas perturbé le fonctionnement du tag, le test suivant consiste à appliquer de l'azote liquide sur le tag. La température de l'azote liquide est d'environ -200 ˚C, son point d'ébullition étant de -195,79 ˚C. Les limites de températures de fonctionnement normales spécifiées dans la documentation de la Mifare Classic sont dès lors très largement dépassées. L'expérience a consisté à tremper un tag RFID de type Mifare Classic dans un Dewar d'azote liquide. L'isolation du Dewar permet une très lente ébullition de l'azote. Celui-ci reste ainsi plus longtemps à l'état liquide. Le but étant bien sûr de laisser le tag assez longtemps dans le Dewar pour que celui-ci soit refroidi à une température approchant celle de l'azote liquide. Une fois à la bonne température, le tag fut mis dans le champ du SCL3711, et les 3 tests de la section 10 ont été réalisés. Ces tests ont été réalisés plusieurs fois avec des durées de mise en contact à l'azote liquide différentes, ce qui a permis de tester plusieurs températures. Il n'a par contre pas été possible de mesurer exactement la température du tag aux moments où les tests ont été réalisés.

12. C'est-à-dire lorsque la bombe réfrigérante est directement mise en contact avec le composant

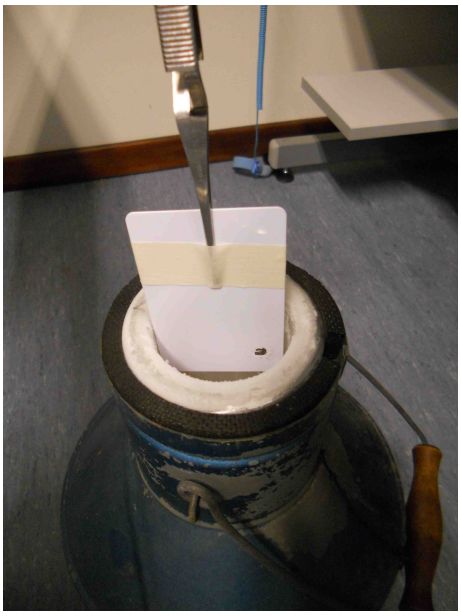

FIGURE 48: Insertion de la Mifare Classic dans le Dewar d'azote liquide

FIGURE 49: Tag Mifare Classic plongé dans de l'azote liquide

Résultats et commentaires

Les différentes durées de mise en contact avec l'azote liquide ont été d' 1, 3 et 10 minutes. Il faut bien sûr noter que le fait de devoir sortir le tag de l'azote pour le positionner dans le champ du lecteur n'est pas idéal pour les résultats car celui-ci a le temps de se réchauffer. C'est pourquoi les différents tests ont été effectués plusieurs fois sur de très courtes périodes de temps (environ 300 nombres générés). La génération des 5000 nombres a été effectuée en plusieurs fois, par pallier de 300 nombres, avec remise dans l'azote liquide de 10 minutes entre chaque génération de 300 nombres. Pour chaque durée d'exposition, les tests de la validité d'un nombre et de la distance entre 2 nombres n'ont présenté aucun résultat négatif. Le test de la graine a lui été effectué sur un tag non isolé de graine non fixe et d'exposition de 10 minutes, dont les résultats sont présentés sur les figures 50 et 51 :

Nombre de nombres différents	4777
Nombre max d'occurrences d'un nombre	3

FIGURE 50: Nombre de nombres différents et maximum d'occurrences de la graine aléatoire d'une Mifare Classic refroidie via de l'azote liquide

Nombre apparu	Nombre d'apparitions
01011001001100110111111010001000	3
10111000010110000100010011100111	3
00110101101100100110111111010101	3
01110000011001011010100101011011	3
00001100011010101001111011001110	2
11100101010111110001111110011001	2
01000111101010101100110001110000	2
01011001010101111101111011111010	2
00001011110110011010000011110010	2
11001110111111001000001000001011	2
00011111010000111010001111100001	2
10000000101000001011010010111001	2
01100110101011011111100001010110	2
11000111011011010110000001111000	2
00110011011111101000100000101101	2
10111100011100000000000100001001	2
11100001011101001110001010110100	2
11101000011000100001100010110101	2
11000001100000011000011110101101	2
10110000001011101011111111111001	2

FIGURE 51: Nombres d'occurrences des 20 nombres pseudo-aléatoires apparaissant le plus de fois pour une graine aléatoire de Mifare Classic refroidie via de l'azote liquide

Ces résultats, lorsqu'ils sont comparés avec ceux de la section 9.1.3 semblent indiquer la présence d'une graine pratiquement aussi aléatoire qu'une graine non soumise à un refroidissement. Comme le montre la figure 50, un nombre n'apparaît en effet que 3 fois au maximum parmi les 5000 nombres. De plus, 4777 nombres sont différents parmi les 5000 obtenus. La figure 51 présente quant à elle des résultats semblables à la figure 32. Les tests des figures 52 et 53 ont également pu être réalisés grâce à la connaissance du générateur de la Mifare Classic.

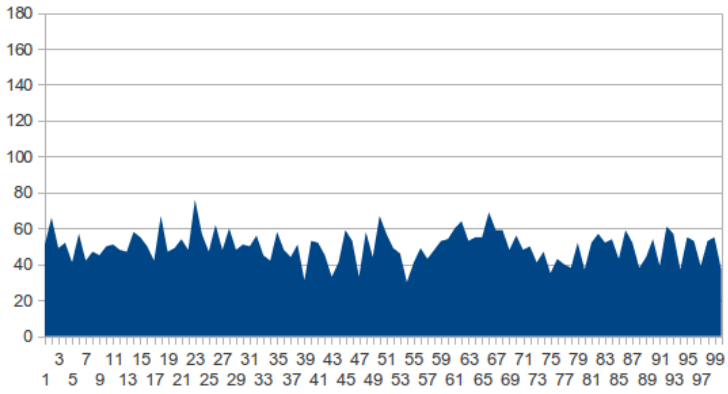

FIGURE 52: Graphique de l'intervalle des index des graines pour une Mifare Classic de graine aléatoire soumise à un refroidissement via de l'azote liquide

Valeur	Interprétation
145.59	Pas aléatoire

FIGURE 53: Résultats et interprétation du test chi-carré pour une graine aléatoire de Mifare Classic refroidie via de l'azote liquide

Le graphique de la figure 52 semble lui aussi très semblable à celui d'un tag non soumis à un événement extérieur tel que présenté à la figure 33. Le test du chi-carré présenté à la figure 53 semble lui imposer un peu plus de vigilance, étant donné qu'il renvoie l'interprétation "Pas Aléatoire". Néanmoins, ce test n'a été réalisé qu'une seule fois étant donné les contraintes appliquées à la génération dans de telles conditions. Il n'est donc pas possible de conclure avec affirmation que le test du chi-carré a échoué et que la graine soumise à de l'azote liquide est "moins aléatoire" qu'une graine non soumise à ces conditions, mais il faut faire preuve de réserve, et une faible influence du froid intense sur la génération de la graine du PRNG de la Mifare Classic n'est pas à exclure.

On peut néanmoins conclure, au vu des trois tests réalisés, que bien que refroidi à -200°C pendant une durée de plusieurs minutes, le tag et plus précisément son générateur, semble avoir un comportement de fonctionnement normal, contrairement à ce que laissait sous-entendre la documentation de celui-ci. Le seul point négatif provient de la génération de sa graine, qui pourrait être très faiblement influencée par ce paramètre. Ces propos sont cependant à nuancer étant donné que le test du chi-carré n'a pu être réalisé qu'une seule fois.

12 Conclusion

Au cours de ce travail, une procédure de test des PRNG implémentés dans la technologie RFID a été établie. Celle-ci a pu être définie à partir des première et seconde parties, qui ont précisé son champ d'application. Les faiblesses présentes dans le PRNG de la Mifare Classic ont réellement constitué une base solide de travail pour la définition et la réalisation des tests de la procédure. D'ailleurs, cette procédure appliquée telle quelle à la Mifare Classic permet de retrouver les différentes faiblesses de son générateur ainsi que d'identifier ce dernier. Cette procédure propose une méthode d'analyse relativement simple en comparaison aux travaux de rétro-ingénierie effectués dans les travaux de Nohl, Plötz, Staburg et Evans [NESP08]. De plus, cette procédure a permis d'apporter des informations supplémentaires sur la graine du PRNG de la Mifare Classic. En effet, les tests réalisés à la section ont permis d'obtenir le résultat suivant : Soit les graines fixes de Mifare Classic différentes sont différentes, soit certains protocoles exécutés avant l'envoi de nombre par le tag prennent plus ou moins de temps, en fonction du tag. Il faudrait imaginer pour la suite un test prenant en compte la vitesse de génération des nombres pour les 3 Mifare Classic.

La procédure, appliquée à deux autres tags de la famille Mifare, à savoir les Mifare DESFire et Ultralight C n'a permis de trouver aucune faiblesse. La période de ces deux tags est plus importante que 1,5 millions. De plus, leurs graines respectives sont vraisemblablement générées aléatoirement. Enfin, leur générateur n'est ni un LFSR simple à 2,4 ou 6 taps, ni un LCG de la liste de la figure 26. Au vu des résultats, ces tags semblent posséder un générateur de nombres pseudo-aléatoires bien plus sûr que celui de la Mifare Classic. Ils ne présentent en tout cas aucune des faiblesses de la Mifare Classic, preuve que la société NXP a fait preuve de plus de prudence lors de la conception du PRNG de ces 2 tags.

Cette procédure n'a pas encore pu être appliquée sur d'autres tags RFID. Cela constitue naturellement une priorité dans une perspective de poursuite de ce travail. Le tag HID IClass, notamment, semble être tout désigné pour une analyse car son générateur a lui aussi révélé ses secrets dans le courant de l'année 2010 [13]. L'application de la procédure à ce tag permettrait sans doute d'élaborer diverses améliorations et changements à appliquer à cette procédure, à l'instar de l'analyse du PRNG de la Mifare Classic.

En ce qui concerne les expériences réalisées sur la Mifare Classic et ayant pour but de perturber le générateur de celle-ci, aucun test pour aucune des expériences n'a produit un résultat sensiblement différent de celui observé dans un environnement normal de fonctionnement. Il serait ici aussi intéressant d'aller plus loin en expérimentant divers autres éléments

13. http ://www.openpcd.org/HID_iClass_demystified

pouvant perturber le générateur du tag. Bien qu'il semble très difficile de soumettre un tag à des températures inférieures, l'échauffement de celui-ci pourrait être réalisé à l'aide, par exemple, d'une étuve ou d'un laser plus puissant.

Références

[AM10] Gildas Avoine and Tania Martin, *Ucl, ingi 2144, rfid business card.*

[AMS08] Gildas Avoine, Tania Martin, and Jean-Pierre Szikora, *Nxp mifare classic : une star déchue*, Misc N°39 (2008).

[Avo] Gildas Avoine, *Ucl, ingi 2144, secured systems engeneering, chapter 13 : Generating randomness.*

[CMB+06] Chamberlain, Mazzara, Blanchard, Musti, Burlingame, Son, Chandramohan, Stump, Forestier, Weiss, and Griffith, *Ibm websphere rfid handbook : A solution guide*, 1 ed., International Business Machines Corporation, 2006.

[Fin10] Klaus Finkenzeller, *Rfid handbook : Fundamentals and applications in contactless smart cards and identification*, 3 ed., John Wiley & Sons, Inc., New York, NY, USA, 2010.

[GdKGM+08] Flavio D. Garcia, Gerhard de Koning Gans, Ruben Muijrers, Peter van Rossum, Ronny Wichers Schreur, and Bart Jacobs, *Dismantling mifare classic*, Institute for Computing and Information Sciences, Radboud University, Nijmegen, The Netherlands (2008).

[GdKM+08] Flavio D. Garcia, Gerhard de Koning Gans, Ruben Muijrers, Peter van Rossum, Roel Verdult, Ronny Wichers Schreur, and Bart Jacobs, *Dismantling MIFARE classic*, 13th European Symposium on Research in Computer Security (ESORICS 2008) (S. Jajodia and J. Lopez, eds.), Lecture Notes in Computer Science, vol. 5283, Springer Verlag, 2008, pp. 97–114.

[Hal04] Haldir, *How to crack a linear congruential generator*, www.reteam.org/papers/e59.pdf, 2004.

[Knu81] Donald E. Knuth, *Seminumerical algorithms*, second ed., The Art of Computer Programming, vol. 2, Addison-Wesley, Reading, Massachusetts, 1981.

[KSF] John Kesley, Bruce Scheiner, and Niels Ferguson, *Yarrow-160 : Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator*, Counterpane Systems ; Minneapolis, USA.

[Lom08] Chris Lomont, *Random number generation*, http ://www.lomont.org/, 2008.

[LS09] Pierre L'Ecuyer and Richard Simard, *Testu01 - a software library in ansi c for empirical testing of random number generators - user's guide, compact version*, Université de Montréal (2009).

[McL09] O McLaughlin, *Mifare desfire specification*, Bracknell Forest Borough Council (2009).

[MHCPL11] Mohamad Merhi, Julio Cesar Hernandez-Castro, and Pedro Peris-Lopez, *Studying the pseudo random number generator of a low-cost rfid tag*, IEEE International Conference on RFID-Technologies and Applications (2011).

[MN98] M. Matsumoto and T. Nishimura, *Mersenne twister : A 623-dimensionally equidistributed uniform pseudorandom number generator*, ACM Trans. on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3-30 (1998) (1998).

[MVO96] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot, *Handbook of applied cryptography*, 1 ed., CRC Press, Inc., Boca Raton, FL, USA, 1996.

[MW06] Rob van Kranenburg Matt Ward, *Rfid : Frequency, standards, adoption and innovation*, Ph.D. thesis, JISC Technology and Standards Watch, 2006.

[NESP08] Karsten Nohl, David Evans, Starbug Starbug, and Henryk Plötz, *Reverse-engineering a cryptographic rfid tag*, Proceedings of the 17th conference on Security symposium (Berkeley, CA, USA), SS'08, USENIX Association, 2008, pp. 185–193.

[NXP09] NXP Semiconductors, *Mf0icu2 - mifare ultralight c*, 2009.

[NXP11a] NXP Semiconductors, *An1303 - mifare ultralight as type 2 tag*, 2011.

[NXP11b] NXP Semiconductors, *Mf1s503x - mifare classic 1k*, 2011.

[PW09] Kyle E. Penri-Williams, *Implementing an rfid 'mifare classic' attack*, Ph.D. thesis, School of Engineering and Mathematical Sciences, London, 2009.

[Rod] François Rodier, *Institut de mathématiques de luminy, cours de cryptographie symétrique, registre à décalage à rétroaction linéaire.*

[RSN+10] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, and San Vo, *A statistical test suite for random and pseudorandom number generators for cryptographic applications*, NIST (2010).

[Ser05] Nicolas Seriot, *Les systèmes d'identification radio (rfid) — fonctionnement, applications et dangers.*

[Suh04] Ahamed Suhile, *Radio frequency identification system (rfid) - rfid 101*, KCP Technologies Limited (2004).

[Tan09] Wee Hon Tan, *Practical attacks on the mifare classic*, Ph.D. thesis, Imperial College London, Department of Computing, 2009.